

ÉCOLE DOCTORALE MIPTIS
*MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE
ET INGÉNIEURIE DES SYSTÈMES*

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE
présentée par :

Abderrahim AIT WAKRIME

soutenue le : **10 Décembre 2015**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **Informatique**

**Une approche par composants pour l'analyse visuelle
interactive de résultats issus de simulations numériques**

THÈSE DIRIGÉE PAR :

Sébastien LIMET

Professeur des Universités, Université d'Orléans, France

Sophie ROBERT

Maître de Conférences, Université d'Orléans, France

RAPPORTEURS :

Joaquín EZPELETA

Professeur des Universités, Université de Saragosse, Espagne

Nabil HAMEURLAIN

Maître de Conférences HDR, Université de Pau et des Pays de l'Adour, France

JURY :

Joaquín EZPELETA

Professeur des Universités, Université de Saragosse, Espagne

Nabil HAMEURLAIN

Maître de Conférences HDR, Université de Pau et des Pays de l'Adour, France

Nicolas FEREY

Maître de Conférences, Université Paris Sud, France

Jean-Michel COUVREUR

Professeur des Universités, Université d'Orléans, France

Sophie ROBERT

Maître de Conférences, Université d'Orléans, France

Sébastien LIMET

Professeur des Universités, Université d'Orléans, France

À ma mère, à mon père, à ma sœur et à mes frères.

REMERCIEMENTS

Une thèse est un travail long, demandant beaucoup d’investissement personnel et surtout parsème de doutes. C’est aussi, avant tout, une expérience humaine à part entière. Cette thèse constitue une riche expérience qui ne peut finir sans remercier les personnes qui m’ont encadré, aidé et soutenu durant ces trois dernières années. Je tiens à remercier l’ensemble des personnes qui ont contribué, de près ou de loin, à l’achèvement de ce travail.

Je tiens à adresser mes plus sincères remerciements à mon directeur de thèse, Sébastien Limet, Professeur à l’Université d’Orléans, pour la confiance qu’il m’a accordée en acceptant de diriger cette thèse. Je le remercie aussi d’avoir été toujours disponible et de m’avoir accordé la liberté dans mes directions de recherche. Je tiens aussi à remercier Sophie Robert, Maître de Conférences à l’Université d’Orléans, de m’avoir encadré durant ces années de thèse. J’ai énormément apprécié les années de travail et l’entière liberté qu’elle m’a accordée tout au long de ces années d’agréable collaboration.

J’adresse mes vifs remerciements à Joaquín Ezpeleta, Professeur à l’Université de Saragosse en Espagne, et Nabil Hameurlain, Maître de Conférences HDR à l’Université de Pau et des Pays de l’Adour en France, pour l’honneur qu’ils m’ont fait d’avoir accepté de rapporter cette thèse. Je les remercie pour la rapidité de lecture de mon manuscrit, malgré un emploi du temps très chargé. J’ai apprécié la profondeur de la relecture et la pertinence de leurs commentaires et de leurs remarques constructives qui ont permis d’améliorer la qualité de cette thèse.

Je tiens à remercier tous les membres de jury qui ont accepté d’évaluer mon travail. Merci à Nicolas Ferey, Maître de Conférences à l’Université Paris Sud en France et Jean-Michel Couvreur, Professeur à l’Université d’Orléans en France, d’avoir accepté d’examiner mon travail. Je remercie Jean-Michel Couvreur de m’avoir accordé l’honneur d’être le président de mon jury.

Un grand merci aux membres de mon équipe PAMDA pour l’accueil et la bonne ambiance et mes collègues du Laboratoire d’Informatique Fondamentale d’Orléans pour les moments agréables qu’on a passés ensemble. Particulièrement merci à Jérôme Durand-Lose directeur du LIFO pour ses conseils et son soutien.

Je tiens ensuite à remercier toutes les personnes, du Pôle Informatique du Collégium Sciences et Techniques à l’Université d’Orléans et de l’IUT d’Orléans, que j’ai eu le plaisir de côtoyer pendant ces années et qui m’ont soutenu d’une façon ou d’une autre.

Je n’oublie pas de remercier tous mes amis pour les moments inoubliables qu’on a pu passer ensemble.

Je remercie avec grande émotion ma famille, en particulier mes parents, ma sœur et mes frères pour son irremplaçable soutien et l’aide qu’ils m’ont toujours apportée.

TABLE DES MATIÈRES

TABLE DES MATIÈRES	vii
LISTE DES FIGURES	x
LISTE DES TABLEAUX	xii
INTRODUCTION GÉNÉRALE	1
APPORT DE LA THÈSE	7
I État de l’art et préliminaires	9
1 LA VISUALISATION SCIENTIFIQUE : HISTORIQUE, CONCEPTS ET TECHNIQUES	11
1.1 INTRODUCTION	11
1.2 ORIGINES, LES DÉFINITIONS ET LES CONCEPTS	12
1.2.1 Origines	12
1.2.2 Définitions	12
1.3 LA VISUALISATION SCIENTIFIQUE ET LA RÉALITÉ VIRTUELLE	14
1.3.1 La Réalité Virtuelle	14
1.3.2 La visualisation scientifique interactive	16
1.4 CONCLUSION	17
2 ARCHITECTURE PAR COMPOSANTS	19
2.1 INTRODUCTION	19
2.2 HISTORIQUE DE LA PROGRAMMATION PAR COMPOSANTS	20
2.3 CONCEPTS DE LA PROGRAMMATION PAR COMPOSANTS	21
2.3.1 Composant logiciel	22
2.3.2 Interfaces	23
2.3.3 Connecteurs	24
2.3.4 Langage de description des architectures par composants	25
2.4 MODÈLES À COMPOSITION SPATIALE	25
2.4.1 Modèles généraux	25
2.4.2 Modèles spécifiques	30
2.4.3 Analyse & Synthèse	33
	vii

2.5	MODÈLES À COMPOSITION TEMPORELLE	35
2.5.1	Workflow	35
2.5.2	Systèmes de workflow scientifique	37
2.5.3	Analyse & Synthèse	45
2.6	MODÈLES À COMPOSITION SPATIO-TEMPORELLE	45
2.6.1	Systèmes de composant logiciel & workflow scientifique	45
2.6.2	Analyse & Synthèse	48
2.7	LA RECONFIGURATION DES ARCHITECTURES PAR COMPOSANTS	48
2.7.1	La reconfiguration centralisée	48
2.7.2	La reconfiguration auto-adaptative	50
2.7.3	Analyse & Synthèse	51
2.8	CONCLUSION	51
3	LA FORMALISATION DES MODÈLES	53
3.1	INTRODUCTION	53
3.2	L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES	54
3.2.1	UML	55
3.2.2	Méthode B	56
3.3	LA SIMULATION ET LE TEST POUR LA VÉRIFICATION	57
3.3.1	Vérification par la simulation	57
3.3.2	Vérification par le test	58
3.4	LES MÉTHODES FORMELLES	58
3.4.1	Définitions	58
3.4.2	Model Checking	59
3.4.3	SAT	61
3.4.4	Les algèbres de processus	61
3.4.5	Les réseaux de Petri	63
3.5	CONCLUSION	64
II	Contributions	67
4	LE CADRE GÉNÉRAL DES CONTRIBUTIONS	69
4.1	INTRODUCTION	69
4.2	MODÈLE DE COMPOSANTS	69
4.3	FORMALISATION DU MODÈLE DE COMPOSANTS	70
4.4	RECONFIGURATION DYNAMIQUE DU MODÈLE DE COMPOSANTS	71
5	LE MODÈLE ComSA	75
5.1	INTRODUCTION	75
5.2	LE MODÈLE ComSA	76
5.2.1	Les composants	76
5.2.2	Les connecteurs	78
5.2.3	Les liens	80

5.3	GRAPHE D'APPLICATION	81
5.4	LA SÉMANTIQUE DU MODÈLE ComSA	83
5.5	CONCLUSION	88
6	LA MODÉLISATION DU MODÈLE ComSA	91
6.1	INTRODUCTION	91
6.2	MODÉLISATION EN RÉSEAUX FIFO COLORÉS	92
6.2.1	Réseaux FIFO Colorés stricts	92
6.2.2	Modèle des composants en sCFN	98
6.2.3	Modèle des connecteurs en sCFN	103
6.2.4	Modèle d'une application en sCFN	106
6.3	L'ANALYSE DES BLOCAGES DES APPLICATIONS ComSA	109
6.3.1	La vivacité des sCFN	110
6.3.2	La configuration de démarrage des applications ComSA	119
6.4	CONCLUSION	122
7	RECONFIGURATION DYNAMIQUE DU MODÈLE ComSA	123
7.1	INTRODUCTION	123
7.2	LA RECONFIGURATION DYNAMIQUE DES APPLICATIONS ComSA	124
7.2.1	Vue générale du processus de la reconfiguration	124
7.2.2	La délimitation de la région de sécurité	127
7.3	LA CORRECTION DE LA RECONFIGURATION DYNAMIQUE DU MODÈLE ComSA	131
7.3.1	Les invariants de la reconfiguration dynamique du modèle ComSA	131
7.3.2	La vérification de la correction de la reconfiguration dynamique du modèle ComSA	132
7.4	CONCLUSION	136
8	VALIDATION ET MISE EN ŒUVRE	139
8.1	INTRODUCTION	139
8.2	LA PLATEFORME ComSATool	139
8.3	ARCHITECTURE DE ComSATool	142
8.3.1	Le vérificateur de la vivacité en sCFN	143
8.3.2	Le reconfigurateur dynamique des applications ComSA	143
8.4	LES DÉTAILS D'IMPLANTATION DE ComSATool	146
8.4.1	Implantation du Vérificateur	146
8.4.2	Implantation du Reconfigurateur	150
8.5	CONCLUSION	151
	CONCLUSION GÉNÉRALE ET PERSPECTIVES	153
	PUBLICATIONS CONNEXES	157
A	REPRÉSENTATION DE LA DESCRIPTION DE L'ARCHITECTURE ET DE COM- PORTEMENTS DES APPLICATIONS ComSA.	161

B	EXEMPLE D'UN FICHIER .DIMACS	167
	BIBLIOGRAPHIE	169

LISTE DES FIGURES

1.1	Aurochs représentés dans la grotte de Lascaux.	12
1.2	L'interaction, l'immersion et l'autonomie en Réalité Virtuelle et la visualisation scientifique interactive ([194]).	16
2.1	L'historique des architectures logicielles.	22
2.2	Un composant CORBA et ses ports.	26
2.3	L'orchestration des services Web.	29
2.4	La chorégraphie des services Web.	29
2.5	Un bloc fonctionnel IEC 61499.	30
2.6	Exemple d'un workflow scientifique.	36
2.7	Exemple d'un composant dans Discovery Net	38
2.8	Exemple d'un DAG avec des nœuds de données et des nœuds de contrôle.	40
3.1	Le processus Y dirigé par les modèles.	55
3.2	Le Principe du Model Checking.	60
3.3	Les modèles principaux de réseaux de Petri [81].	64
4.1	Les modèles principaux de réseaux de Petri avec les réseaux FIFO colorés stricts représentés avec la couleur rouge.	71
4.2	Le processus d'étude de la vivacité des applications ComSA.	72
4.3	Le processus de reconfiguration des applications ComSA.	72
5.1	Exemple d'un composant avec ses relations d'incidence.	77
5.2	Le comportement du composant du modèle ComSA.	79
5.3	Les connecteurs : (a) sFIFO, (b) bBuffer, (c) nbBuffer, (d) bGreedy et (e) nbGreedy.	80
5.4	Exemple de graphe d'application (seuls deux liens déclencheurs ont été représentés pour ne pas charger le graphe).	82
5.5	L'illustration de la sémantique des composants et des connecteurs ComSA.	87
6.1	Les différents éléments des réseaux sCFN.	93
6.2	Exemple d'un réseau sCFN et son évolution.	95

6.3	(a) représente un sCFN et (b) son CPN équivalent.	97
6.4	Structures sCFN de : (a) la duplication, (b) la synchronisation et (c) l'itération.	98
6.5	Représentation du composant de la remarque 1.	99
6.6	$sCFN_C(C)$ pour le composant de la remarque 1. Les places l_1, l_2, l_3, l_4 ainsi que les transitions $t'_{i_1}, t'_{i_2}, t'_{o_1}$ et t'_{o_2} ne font pas partie de $sCFN_C(C)$ mais représentent comment les ports du composant sont reliés à l'extérieur.	102
6.7	sCFN du connecteur sFIFO.	104
6.8	Réseau FIFO coloré strict du connecteur bBuffer (a) et le connecteur nbBuffer (b).	104
6.9	Réseau FIFO coloré strict du connecteur bGreedy (a) et le connecteur nbGreedy (b).	105
6.10	Exemple d'un siphon E contenant une trappe Q	112
6.11	L'illustration d'une transition couvrante et d'une transition non couvrante.	113
6.12	(a) Le réseau Lossy (état initial), (b) étape 2 du réseau Lossy de la figure (a), (c) étape 3 du réseau Lossy de la figure (a) et (d) étape 4 du réseau Lossy de la figure (a).	115
6.13	(a) Exemple d'un sCFN (état initial), (b) étape 2 du sCFN de la figure (a), (c) Simulation du sCFN de la figure (a) et (d) Simulation du sCFN de la figure (b).	116
6.14	Un exemple d'un réseau ACN et d'un réseau non ACN.	118
6.15	Graphe d'une application de dynamique moléculaire.	119
6.16	Les différentes étapes du processus de démarrage d'une application ComSA.	121
7.1	Les différents états de la reconfiguration.	125
7.2	L'illustration des différentes étapes de l'algorithme 3	130
7.3	La région de sécurité impactée par la reconfiguration.	131
7.4	La région de sécurité impactée par la suppression du connecteur nbG_8 de la figure 6.15.	133
8.1	L'interface graphique principale de la plateforme ComSATool.	140
8.2	L'interface graphique des sCFN de la plateforme ComSATool.	141
8.3	L'architecture de la plateforme ComSATool.	142
8.4	L'architecture du vérificateur de la plateforme ComSA.	144
8.5	L'architecture du reconfigurateur de la plateforme ComSA.	145
8.6	Le sCFN du graphe d'application de l'exemple de la figure 6.15.	147
8.7	Le fichier <code>.scfn</code> définissant le sCFN de la figure 6.6.	148
8.8	Un fichier <code>.dimacs</code> exemple avec 4 variables et 3 clauses.	148
8.9	L'évaluation de la performance de l'étude de la propriété "Trappes \in Siphons" par rapport au nombre de places.	150

Liste des tableaux

2.1	Comparaison des modèles CCM, FRACTAL, services Web, IEC 61499, BIP, L2C et CCA.	34
5.1	Les états des connecteurs de type Greedy.	85
5.2	Les états des connecteurs de type Buffer.	86
6.1	Les siphons contenant une trappe détectés de l'exemple illustré par la figure 6.15	122
8.1	L'évaluation de la performance du module <i>Vérificateur</i> de l'approche ComSA.	149

INTRODUCTION GÉNÉRALE

Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better.

Donald Knuth

De nos jours, les simulations sont devenues incontournables pour la modélisation des systèmes naturels dans divers domaines comme : physique [14], aéronautique [219], biologie [80], physique nucléaire [62], sciences sociales [30], etc. Les simulations numériques diminuent les risques et les dangers provenant d’une simulation réelle. Elles sont de plus en plus utilisées pour simuler des phénomènes physiques réels et croissent en nombre et en complexité. Cette complexité croissante est accentuée par le développement et l’utilisation de nouvelles technologies. Une simulation scientifique offre des gains de temps et de production très importants pour les scientifiques.

Les bénéfices fournis par une simulation informatique sont dus à l’évolution des ordinateurs et aux nouvelles générations de processeurs. Les processeurs d’aujourd’hui ont connu un grand progrès spécialement au niveau de la haute performance grâce à une fréquence de fonctionnement élevée et à l’exécution d’opérations parallèles (parallélisme).

Pour augmenter la puissance de calcul pour les simulations scientifiques, les scientifiques peuvent utiliser une grappe de serveurs qui est une ferme de calcul qui se compose de plusieurs ordinateurs regroupés pour les gérer comme une seule ressource informatique. Cette grappe est utilisée pour dépasser les capacités d’un ordinateur et est composée dans la plupart des cas d’environnements répartis sur plusieurs sites et reliés par le réseau, avec une puissance et une capacité de stockage importantes.

De plus, avec l’essor du rendu graphique, le rôle de l’image est devenu primordial. Grâce au développement des cartes graphiques, les capacités de visualisation se sont développées prodigieusement. Ce progrès a donné naissance à une nouvelle discipline scientifique appelée *visualisation scientifique*. Associée à la simulation scientifique, la visualisation et particulièrement la *visualisation interactive* devient un outil essentiel pour bien observer, analyser et comprendre les phénomènes complexes étudiés dans plusieurs disciplines scientifiques comme les nano-sciences où le projet *ExaviZ*¹ (Exa-

1. <http://exaviz.simlab.ibpc.fr/>

scalable visual analysis for life & materials sciences) s'inscrit. Ce projet vise à étudier la matière à l'échelle nanoscopique particulièrement au niveau des simulations de dynamique moléculaire haute performance ayant comme objectif l'étude des complexes moléculaires. Cette nouvelle étude vise à concevoir des applications de simulations afin d'effectuer le post-traitement, l'analyse, la visualisation, l'exploration et l'interprétation des données générées.

Néanmoins, avec la croissance du besoin d'utilisation des applications de simulations et visualisation scientifique, la compréhension des phénomènes étudiés nécessitent souvent d'avoir des connaissances de ces disciplines scientifiques. La conception de ces applications devient très difficile pour les scientifiques, particulièrement les non informaticiens.

Pour faciliter le couplage des différents codes de calculs, de visualisation et d'exploration souvent hétérogènes et difficiles à faire coopérer, les scientifiques doivent être capables d'utiliser et de réutiliser les différentes parties de l'application déjà construites afin de pouvoir interchanger certaines parties. Ainsi, une nouvelle méthode de conception, d'intégration et d'interopérabilité qui a de bonnes propriétés basée sur le paradigme de *programmation par composants* est utilisée.

Les architectures par composants visent à fournir des méthodes pour l'abstraction des différentes parties des applications à développer par des composants. Ces derniers communiquent entre eux grâce à un couplage et devraient être faciles à recomposer, pour répondre aux nouveaux besoins des développeurs en mettant en œuvre des abstractions relatives à une classe de problèmes particulière. D'autre part, la complexité des techniques mises en jeu fait apparaître la nécessité de factoriser les efforts de développement, donc l'approche modulaire des architectures basées sur la notion de composants tente d'y répondre.

La programmation par composants a pour vocation d'offrir plusieurs avantages par rapport aux autres méthodes comme la programmation par objets. L'un de ces avantages est la réutilisation et l'exploitation des composants pré-construits. Ce sont des éléments constitutifs d'un logiciel destinés à être incorporés en tant que blocs ou modules composables pour interagir en échangeant des messages. Les composants sont des éléments architecturaux qui encapsulent la partie métier de l'architecture et possèdent une partie externe représentant un composant qui contient des interfaces. Elles décrivent la spécification des fonctionnalités, les services fournis et les services requis par le composant. De plus, les composants favorisent l'autonomie, l'incorporation dans les applications tierces et quelquefois l'indépendance des langages de programmations utilisés dans le but de faciliter l'intégration d'applications en passant par l'échange des données. Un administrateur système, un intégrateur de système ou un développeur d'applications peut avoir besoin de construire un système ou un service à partir de composants existants, soit pour répondre à des défaillances, dans le cadre de l'évolution d'un système, ou pour introduire de nouvelles applications dans un système en fonctionnement. Dans ce cas, nous parlons de *la composition des composants de l'application*.

La composition des composants de l'application est l'opération qui consiste à as-

sembler des composants pré-développés et pré-testés pour produire ou enrichir des applications. Le résultat de la composition est une application complexe au niveau architectural répondant aux besoins de l'utilisateur.

Dans cette thèse, nous nous intéressons à la formalisation et la dynamique des applications de visualisation scientifique interactives, notamment dans le cadre du projet *ExaviZ*. Ce dernier a pour but l'exploration interactive de très gros volumes de données issus de simulations numériques et de l'analyse conjointe de résultats expérimentaux dans les sciences du vivant ou des matériaux. Le type d'applications étudié s'inscrit dans le domaine de l'*analyse visuelle* [214] qui se concentre sur le raisonnement analytique facilité par des interfaces visuelles interactives. Ce domaine combine le développement des technologies informatiques et l'intelligence humaine pour mieux analyser et comprendre les problèmes complexes provenant de la production rapide en masse de données.

Ces applications nécessitent une contrainte essentielle telle que la performance, par exemple le taux d'images par seconde, alors que les différents éléments de ces applications peuvent fonctionner à des fréquences très différentes.

La performance dans les modèles par composants décrits dans la littérature, surtout les propriétés de synchronisation, n'est pas fournie pour les applications de visualisation scientifique interactive [26, 104]. D'autres modèles existants se heurtent aussi au problème de synchronisation entre les composants au sein d'une application à savoir favoriser la désynchronisation des composants qui ont des fréquences d'exécution très différentes [50, 105]. La considération et la garantie de la désynchronisation de composants, qui ont des fréquences d'exécution très différentes, permettent d'obtenir des performances intéressantes dans le cadre d'une application interactive.

Il est donc nécessaire de définir un modèle d'assemblage répondant spécifiquement aux caractéristiques de visualisation scientifique. Là encore, les approches par composants jouent un rôle important dans la séparation des codes fonctionnels et des codes non fonctionnels permettant de spécifier plus facilement des schémas de communications pour gérer et gouverner les interactions entre les composants. Par exemple, les applications de visualisation scientifique interactives peuvent impliquer des contraintes sur la cohérence de données face à des schémas de communications basés sur la perte de données pour assurer la performance de l'application. Il est donc nécessaire de maîtriser ses pertes de données pour assurer la cohérence, par exemple, de deux composants de simulations qui coopèrent pour un troisième composant de calcul ou pour l'affichage des résultats dans un composant de rendu graphique.

Dans ce contexte, une approche par composants prenant en compte les caractéristiques précises des applications de visualisation scientifique interactives est proposée. Dans cette thèse, nous nous intéressons précisément à la définition d'une approche basée sur une *coordination exogène* afin de représenter le cas général des applications visées. La composition des composants pour construire une application pose plusieurs problèmes. Parmi ceux-ci, il y a deux aspects : (1) garantir une composition bien définie afin d'assurer le démarrage d'une application, prévenir les problèmes de blocage

durant l'exécution et (2) fournir une reconfiguration performante en modifiant dynamiquement l'architecture d'une application sans l'arrêter complètement.

L'étude du démarrage d'une application interactive basée sur une architecture par composants revient à étudier sous quelle condition l'application en question peut se lancer et s'exécuter proprement. Également, cette étude permet de prévenir les éventuels blocages qui peuvent se produire lors de l'exécution. Ceci implique de résoudre le problème de détection des blocages qui a été étudié par plusieurs travaux récents [42, 136, 146, 221].

L'émergence rapide des architectures par composants a fait apparaître certaines difficultés limitant leur évolution. En effet, la reconfiguration est abordée afin d'intégrer les aspects dynamiques comme le lancement ou l'arrêt à la volée de composants pour faire face à des évolutions de leur contexte de fonctionnement et de leur utilisation. La gestion de la reconfiguration dynamique des applications de visualisation scientifique interactives doit tenir en compte des spécificités de ce type d'applications haute performance. Cette gestion permet de modifier la structure d'une application en remplaçant, ajoutant ou supprimant un ou plusieurs de ses éléments. L'objectif est de reconfigurer dynamiquement et à n'importe quel moment une application pendant son exécution sans l'arrêter complètement pour garantir le maximum des services fournis. Le principe de la reconfiguration dynamique proposée évite la reconfiguration classique arrêtant toute l'application avant de réaliser une opération de reconfiguration.

Par conséquent, les problèmes et les aspects qu'il faut traiter sont divers. Dans un premier temps il faut définir un modèle par composants pour satisfaire les contraintes et modéliser des applications de visualisation scientifique interactives prenant en considération leurs caractéristiques par exemple la perte de données. Dans un deuxième temps, proposer et définir une formalisation pour le modèle présenté. Cette formalisation, basée sur les réseaux de Petri, est principalement une sous classe des réseaux FIFO. Par ailleurs, la formalisation sémantique offre une modélisation claire et puissante des comportements d'applications de visualisation scientifique interactives haute performance. Cette classe est utilisée pour simuler et tester, par exemple, la construction et le comportement global d'une application. Une telle modélisation sert également à préciser certaines propriétés structurales de l'application dans le but de garantir son démarrage et son bon fonctionnement sans blocage. Enfin, le dernier problème auquel nous nous intéressons est celui de la reconfiguration dynamique de l'architecture d'une application. Elle permet d'offrir la possibilité de modifier la structure tout en garantissant la disponibilité de l'application et en gardant le maximum de services offerts par les composants.

Des mécanismes et des algorithmes ont été développés afin de réaliser et mettre en œuvre ces objectifs et les solutions apportées pour les problématiques abordées.

En résumé, l'objectif de la thèse est de proposer un modèle par composants permettant de modéliser et analyser les comportements dynamiques des applications de visualisation scientifique interactive. La thèse va donc répondre aux trois questions majeures concernant la définition d'une approche par composants spécifique qui

permettra de bien concevoir les applications interactives contenant des phases dynamiques d'analyse :

- Comment définir une formalisation et une extension de l'**approche par composants** [140] permettant de prendre en compte l'impératif de **performance** des **applications parallèles distribuées**, l'**hétérogénéité** des codes et des plateformes de déploiement ?
- Comment garantir une **composition correcte** pour assurer un démarrage propre d'une application et prévenir les problèmes de **blocage** durant son exécution ?
- Comment réaliser une **reconfiguration dynamique** dédiée aux **applications de visualisation scientifique interactive** sans les arrêter complètement ?

L'objectif de cette thèse est d'identifier et de proposer une approche par composants intégrant des aspects dynamiques comme le lancement ou l'arrêt à la volée de composants. Pour cela nous nous sommes intéressés à tous les aspects énumérés ci-dessus.

APPORT DE LA THÈSE

Les apports de cette thèse sont essentiellement la proposition d’une approche formelle pour la spécification, la vérification et la reconfiguration des applications de visualisation scientifique interactives à base de composants. Les principales contributions de nos travaux sont les suivantes :

- La première contribution consiste à définir et proposer une sémantique formelle basée sur les réseaux de Petri. Le modèle formel proposé permet de modéliser les différents objets composant l’architecture de nos applications à savoir les composants, les connecteurs et les liens. En même temps, le formalisme proposé offre une modélisation claire des différents comportements dynamiques des composants hétérogènes mettant en œuvre la simulation ou les traitements de données, la visualisation et l’interaction.
- La deuxième contribution consiste à trouver des solutions aux problèmes de blocages de ces applications. La sémantique déjà proposée a été utilisée pour étudier la vivacité dans le but d’aider les utilisateurs à concevoir des applications pouvant démarrer correctement et qui ne contiennent pas de blocages.
- La troisième contribution vise à définir et décrire une méthode de reconfiguration dynamique pour les applications interactives dynamiques à base de composants qui ne sont pas persistants et qui ont besoin d’un démarrage ou d’un arrêt à la volée. Nous avons mis en place plusieurs opérations pour modéliser et réaliser la reconfiguration proposée et nous avons utilisé notre formalisme pour vérifier la correction des différentes étapes pour insérer, supprimer ou modifier un composant ou connecteur dans une application.

Pour valider notre démarche, nous présentons un outil qui permet d’aider les utilisateurs et surtout les non informaticiens à créer leurs applications qui sont représentées par des graphes d’applications. L’outil fournit aux utilisateurs plusieurs interfaces graphiques pour modéliser, vérifier la vivacité et reconfigurer les applications construites.

ORGANISATION DU MÉMOIRE

Ce mémoire est découpé en deux grandes parties.

Partie I : État de l'art et préliminaires Le chapitre 1 présente l'historique, les concepts et les techniques de visualisation scientifique avec le positionnement de cette dernière par rapport la réalité virtuelle. Le chapitre 2 introduit en détail les définitions à la compréhension des architectures par composants en termes de concepts utilisés, les différents modèles à composition et leur reconfiguration. Le chapitre 3 est consacré aux méthodes formelles afin de justifier notre choix de la méthode formelle utilisée dans le travail présenté. Ces trois chapitres servent de base à l'ensemble du manuscrit et traitent les préliminaires et l'état de l'art.

Partie II : Contributions Le chapitre 4 présente le cadre général de notre travail en termes du modèle par composants dédié aux applications de visualisation scientifique interactive, la méthode formelle pour le modéliser et la reconfiguration dynamique des applications construites et en cours d'exécution. Le chapitre 5 présente en détail notre modèle par composants, la sémantique de ses différents éléments et comment les assembler pour construire des applications interactives hautes performance. Le chapitre 6 présente notre formalisme basé sur les réseaux de Petri sur lequel est basé nos travaux. Le formalisme proposé permet de modéliser les applications interactives basées sur des composants et comment vérifier leur vivacité en définissant une configuration de leur démarrage. Le chapitre 7 détaille les principales techniques de reconfiguration dynamique des applications de visualisation scientifique interactives et montre comment vérifier la correction de la reconfiguration dynamique étudiée en utilisant la méthode formelle présentée dans le chapitre 6. Finalement, le chapitre 8 détaille l'implémentation et la mise en œuvre de notre modèle en proposant un outil qui permet de modéliser, vérifier et reconfigurer les applications de visualisation scientifique interactives. D'abord, ce chapitre présente l'architecture global de l'outil. Puis, il présente les détails d'implantation des différentes parties de cet outil.

Après la partie I et la partie II, nous concluons ce manuscrit en présentant les apports de notre travail de la thèse, ainsi que les perspectives de recherches ouvertes et envisagées.

Première partie

État de l'art et préliminaires

LA VISUALISATION SCIENTIFIQUE : HISTORIQUE, CONCEPTS ET TECHNIQUES

Study the science of art and the art of science.

Leonardo DA VINCI

SOMMAIRE

1.1	INTRODUCTION	11
1.2	ORIGINES, LES DÉFINITIONS ET LES CONCEPTS	12
1.2.1	Origines	12
1.2.2	Définitions	12
1.3	LA VISUALISATION SCIENTIFIQUE ET LA RÉALITÉ VIRTUELLE	14
1.3.1	La Réalité Virtuelle	14
1.3.2	La visualisation scientifique interactive	16
1.4	CONCLUSION	17

1.1 INTRODUCTION

La visualisation scientifique est une approche contemporaine et éprouvée basée sur la combinaison de l'art de l'intuition humaine et la science de la déduction mathématique pour percevoir directement des représentations visuelles et d'en tirer des connaissances et la perspicacité.

La visualisation scientifique est aussi la formation de métaphores visuelles abstraites. En combinant cette dernière avec la capacité de l'œil humain à discerner des relations en représentant des données complexes sous forme graphique, par exemple, de résultats scientifiques. Ces représentations rendent les données spatio-temporelles compréhensibles et permettent d'en extraire de la connaissance.

Dans ce chapitre nous précisons les termes techniques et les concepts liés à la visualisation scientifique. Ce chapitre est divisé en trois sections. La première section présente le cadre historique, les définitions et les concepts liés à la visualisation scientifique. La deuxième section introduit la relation entre la Réalité Virtuelle et la visualisation scientifique et présente les avantages de l'interaction pour la visualisation scientifique. En plus, la définition de la visualisation scientifique interactive est introduite. Enfin la troisième section conclut le chapitre.

1.2 ORIGINES, LES DÉFINITIONS ET LES CONCEPTS

1.2.1 Origines

La visualisation scientifique, telle qu'elle est actuellement étudiée et pratiquée, est encore une discipline relativement nouvelle. Cette discipline s'est développée pour illustrer graphiquement des données scientifiques pour permettre aux scientifiques de comprendre, d'illustrer et de construire un aperçu de leurs données.

Historiquement, la visualisation est apparue avant l'écriture, aux temps immémoriaux, sous forme d'art graphique, figure 1.1, à savoir les dessins, la typographie, les peintures, la calligraphie, etc. Le dessin fût donc, durant des générations et des siècles, un moyen important de communication et de transmission de l'information à l'exemple des égyptiens en 2500 avant J.-C. En effet, ils utilisaient des rouleaux de papyrus pour écrire et partager leurs pensées avec autrui notamment avec les hiéroglyphes. En 1440, grâce à l'invention de l'imprimerie par Johannes Gutenberg, la production de masse de textes et d'art graphique a été simplifiée et est rapidement devenue un substitut aux transcriptions manuelles.



FIGURE 1.1 – Aurochs représentés dans la grotte de Lascaux.

Avec les ordinateurs d'aujourd'hui, les outils classiques de dessin ont nettement évolué, à l'exemple du pinceau qui a été remplacé par de nouveaux outils comme l'écran tactile. Ce développement a favorisé l'utilisation de logiciels permettant la production de visualisation avec un traitement des grandes masses de données. Le volume de données généré donne lieu à l'apparition de nombreux outils impliquant une visualisation d'une nouvelle génération (2D ou 3D) pour effectuer des analyses.

La visualisation scientifique permet non seulement une meilleure représentation du monde, mais elle permet aussi l'émergence de nouvelles informations difficilement observables autrement.

Parallèlement à l'évolution de la visualisation scientifique, plusieurs définitions et utilisations ont été proposées. Par la suite, nous allons présenter une définition et les concepts liés à la visualisation scientifique.

1.2.2 Définitions

La visualisation scientifique est l'un des domaines de l'informatique qui a le plus évolué ces dernières années. La visualisation scientifique est devenue un axe de re-

cherche important [148] et un nouveau domaine passionnant de la science informatique. Elle est stimulée en grande partie par l'augmentation de la puissance des ordinateurs ainsi qu'au progrès de la technologie des périphériques notamment par la croissance rapide des processeurs CPU (Central Processing Unit) mais également des processeurs graphiques GPU (Graphics Processing Unit).

Dans la littérature, nous trouvons plusieurs définitions de la visualisation scientifique comme dans [24, 97, 173], voici celle proposée par B. McCormick, T. Defanti et M. Brown dans [148] :

« La visualisation est une méthode de calcul. Elle transforme le symbolique dans le géométrique, permettant aux chercheurs d'observer leurs simulations et leurs calculs. La visualisation offre une méthode pour voir l'invisible. Elle enrichit le processus de découverte scientifique et favorise des idées profondes et inattendues. Dans de nombreux domaines, elle est déjà en train de révolutionner la façon dont les scientifiques font de la science. »

La visualisation permet à un scientifique de convertir des données complexes en une représentation visuelle et d'appréhender de manière synthétique et pertinente un ensemble de données sous forme d'images. Ces images sont créées par des ordinateurs dans le but de comprendre les simulations de phénomènes précis où chaque image représente un vecteur d'informations très important. Il faut savoir que la visualisation scientifique a trois objectifs :

- La création d'images expressives et stylisées pour la compréhension et la communication : la visualisation scientifique est utilisée pour désigner toute technique impliquant la transformation de données spatio-temporelles en information visuelle, en utilisant un processus reproductible. Elle caractérise la technologie de l'utilisation de techniques d'infographie pour explorer les résultats de l'analyse numérique, principalement multi-dimensionnelles. Traditionnellement, le processus de visualisation consiste à filtrer les données brutes pour sélectionner une résolution et la région d'intérêt, de la cartographie souhaitée qui résultent en une forme graphique, et de produire une image, animation, ou autre produit visuel ;
- L'analyse et la connaissance de résultats : après la transformation de données complexes en produit visuel, les scientifiques ont la possibilité d'étudier et d'expliquer les phénomènes étudiés. En effet, la représentation visuelle de ces données est souvent indispensable pour acquérir une compréhension du processus impliqué grâce à des raisonnements spatiaux et l'œil humain ;
- L'intégration de la visualisation au système fournissant de données : la visualisation peut être utilisée comme un moyen pour faciliter la compréhension des résultats de simulations pouvant être complexes. Elle est utilisée soit comme une étape intermédiaire ou soit comme une étape finale au niveau des systèmes générant un grand volume de données pour leur compréhension, leur interprétation et leur exploration. Cependant, la visualisation n'est qu'un moyen, et

non une finalité et la réflexion humaine reste indispensable pour comprendre la nature profonde de ces données.

1.3 LA VISUALISATION SCIENTIFIQUE ET LA RÉALITÉ VIRTUELLE

1.3.1 La Réalité Virtuelle

1.3.1.1 Définitions

La Réalité Virtuelle est une simulation informatique interactive immersive, visuelle, sonore et/ou haptique d'environnements réels ou imaginaires. C'est un environnement permettant de simuler la présence physique dans un monde réel ou imaginaire. Son objectif est de faire concevoir à l'utilisateur un monde artificiel ressemblant à un monde réel afin de lui donner la possibilité d'y interagir intuitivement et naturellement. L'intérêt est de mettre l'être humain dans un environnement contrôlé, par exemple, dans un environnement qui serait impossible à reproduire dans le monde réel ou qui serait trop onéreux ou trop risqué.

Dans la littérature, nous trouvons plusieurs définitions différentes mais qui se rejoignent sur un nombre important de mots ou de notions clés. En 1992, les auteurs de [29] ont suggéré une définition de la Réalité Virtuelle et l'ont présenté comme étant une technique de visualisation et d'interaction en exploitant des appareils informatiques et des données diversifiées, complexes et hétérogènes.

Une autre définition considère que la Réalité Virtuelle est une extension des interfaces Homme-Machine [132]. Cette définition est aussi adoptée dans un autre travail qui a démontré que les interfaces simulent des environnements réels et permettent d'effectuer des interactions. Ainsi, *Stephen Ellis* [86] définit la Réalité Virtuelle comme :

« *VR is an advanced human-computer interface that simulates a realistic environment and allows participants to interact with it* »

Durant des années, plusieurs définitions fonctionnelles, techniques, pratiques et philosophiques de la Réalité Virtuelle ont été proposées. Malgré cette diversité, son utilité et sa finalité restent les mêmes. En effet, nous adoptons la définition de [91]

« *La Réalité Virtuelle est un domaine scientifique et technique exploitant l'informatique et les interfaces comportementales en vue de simuler dans un monde virtuel le comportement d'entités 3D, qui sont en interaction en temps réel entre elles et avec un ou des utilisateurs en immersion pseudo-naturelle par l'intermédiaire de canaux sensori-moteurs* »

1.3.1.2 Les composantes de la Réalité Virtuelle

Historiquement, les premières recherches concernant la Réalité Virtuelle ont été concentrées sur le développement et l'implémentation d'interfaces favorisant l'immersion des utilisateurs et l'interaction au niveau des environnements virtuels [91]. L'auteur de l'article [220] a montré que la Réalité Virtuelle se constitue de trois composantes basiques, qui se retrouvent dans la plupart des travaux de ce domaine, qui sont *l'autonomie*, *l'interaction* et *l'immersion*.

Jacques Tisseau a défini une application de la Réalité Virtuelle comme étant une composante contenant deux sous-composantes à savoir la présence et l'autonomie. La composante présence regroupe et intègre les deux composantes interaction et immersion. Ainsi, Tisseau représente ces composantes principales sous forme de points dans un repère à trois dimensions où chaque dimension reflète l'autonomie, l'interaction ou l'immersion (voir figure 1.2) :

- **Autonomie** : est une façon de rendre les environnements plus réactifs. Elle représente une mesure qualitative de la capacité d'un modèle d'agir et de réagir à des événements simulés. L'utilisateur représente une entité principale et il peut fixer les entrées et les paramètres de sa simulation pour étudier et analyser les résultats avec ou sans interactions au cours de l'exécution de sa simulation. En effet, il est représenté au sein de son environnement virtuel, par un avatar, pour entrer en interaction comportementale adaptée et mieux coordonner ses actions et ses comportements [194] ;
- **Interaction** : dans un environnement virtuel, les images et les capacités d'interaction sont améliorées grâce à un traitement spécial des modalités d'affichage non visuelles, notamment auditives et haptiques, afin de convaincre les utilisateurs de leur immersion dans un espace de synthèse. Dans ces environnements, les utilisateurs ont la possibilité d'utiliser leurs yeux, oreilles et mains comme ils le feraient normalement dans le monde réel pour effectuer des interactions virtuelles [86, 93, 172]. L'interaction est la composante qui permet aux utilisateurs d'interagir avec leur environnement, c'est donc une composante *motrice*. Les différents types d'interaction sont :
 - Navigation ;
 - Interaction avec les objets du monde virtuel ;
 - Contrôle d'application.
- **Immersion** : permet de plonger l'utilisateur dans un environnement réel. Il a l'impression d'être véritablement dans un monde réel et ne fait plus la différence avec la vie réelle, ceci à l'aide de mécanismes occultant tout ou partie du monde réel [52, 59]. L'immersion est toujours liée à la présence d'un utilisateur dans son monde virtuel. La présence joue un rôle très important pour avoir une meilleure immersion. En effet, la présence permet à l'utilisateur de satisfaire un sentiment d'être à l'intérieur de l'environnement virtuel.

1.3.2 La visualisation scientifique interactive

La visualisation est un axe très important de la Réalité Virtuelle. L'interaction a rendu la visualisation, précisément la visualisation scientifique, plus expressive. Par conséquent, l'utilisateur peut rester en interaction avec ses calculs scientifiques et peut diriger la simulation dans le but de gagner un nouvel aperçu. Selon le psychologue *Gibson* [99], l'interaction est liée à deux concepts qui sont l'affordance, c-à-d la potentialité, et la perception. Le premier désigne les propriétés actionnables entre le monde et un individu. Le deuxième est inséparable de l'action car il faut agir pour percevoir et inversement. Ce couplage entre l'action et la perception aide les scientifiques à mieux comprendre et analyser les données résultantes de la visualisation.

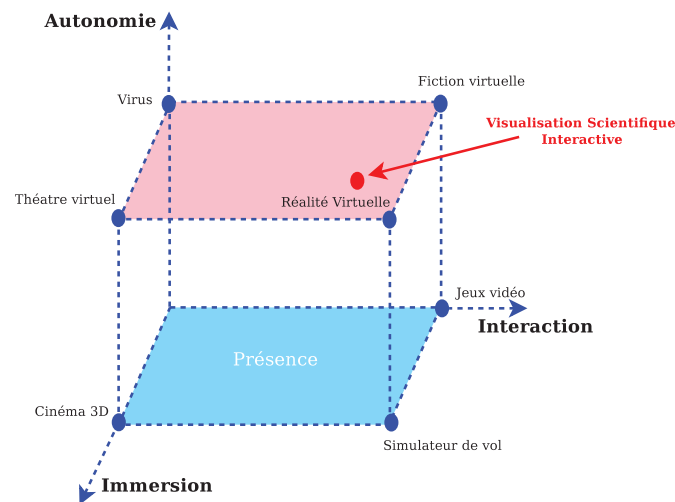


FIGURE 1.2 – L'interaction, l'immersion et l'autonomie en Réalité Virtuelle et la visualisation scientifique interactive ([194]).

La visualisation scientifique interactive, est tout d'abord un moyen pour mieux comprendre un phénomène et de modifier ou améliorer chaque simulation avec moins d'immersion et d'autonomie que la Réalité Virtuelle (voir figure 1.2). Avant de décrire plus en détail les propriétés de la visualisation scientifique interactive, nous donnons sa définition telle qu'elle est énoncée par *S. K. Card*, *J. D. Mackinlay* et *B. Shneiderman* dans [60] :

« *The use of computer-supported, interactive, visual representations of abstract data to amplify cognition* »

Cette définition reste valable pour les deux sous-catégories de la visualisation. La première est la visualisation d'informations et la deuxième est la visualisation scientifique. Principalement, les deux sous-catégories ont été différenciées à l'aide de l'axe d'application s'il est scientifique alors nous parlons de la visualisation scientifique, sinon nous parlons de la visualisation d'informations.

La très grande variété et richesse des définitions existantes de la visualisation scientifique et particulièrement la visualisation scientifique interactive montre son intérêt

et son utilisation dans plusieurs disciplines. Dans le cadre des travaux effectués dans cette thèse, nous proposons et nous adoptons la définition suivante :

Définition 1 (Visualisation scientifique interactive) *La visualisation scientifique interactive est une représentation visuelle de données scientifiques complexes couplée à un système interactif pour interagir avec et comprendre leur signification.*

La visualisation scientifique interactive s'intéresse aux techniques permettant aux scientifiques d'extraire des connaissances à partir des résultats des simulations et des calculs. Elle permet de traduire par une approximation proche à la réalité la possibilité d'acquérir de nouvelles connaissances et de la compréhension de la complexité de calcul scientifique. Les données générées sont transmises aux scientifiques de telle sorte que leur utilisation efficace peut être faite en fonction des capacités analytiques humaines. La manipulation de la représentation visuelle des données générées se réalise, soit au cours de la simulation soit en phase de post-traitement avec une interaction, par la vision humaine et les principes psychologiques de la perception.

Les avantages attendus de la visualisation scientifique interactive sont une analyse plus rapide et plus large, une meilleure compréhension spatiale et de nouveaux moyens d'exploration, de manipulation et d'analyse [133]. En résumé il y a :

- exploration et/ou exploitation des données scientifiques ;
- acquisition de nouvelles connaissances ;
- contrôle de la qualité des simulations et des mesures ;
- modification des valeurs des paramètres et avoir la visualisation de données résultantes en temps réel.

Les applications de visualisation scientifique interactives [127, 138, 211] nécessitent des compétences afin de les concevoir et les développer. La plupart des applications interactives sont basées sur des simulations en plusieurs domaines comme la dynamique des fluides numérique, la biologie, la biochimie ou les géosciences. Dans la science de la vie, les applications interactives sont basées sur les simulations de dynamique moléculaire pour étudier des complexes moléculaires. Ces applications étudiées contiennent trois parties fonctionnelles, notamment *l'interaction*, la *simulation* et la *visualisation*, qui sont très hétérogènes. La partie interaction comporte des périphériques, par exemple un Omni Phantom[®], envoyant des actions vers la partie simulation. Cette dernière permet de faire tourner des calculs pour générer des données pouvant être affichées par la partie visualisation.

1.4 CONCLUSION

Dans ce chapitre, Nous avons introduit le contexte de notre travail en présentant, dans un premier temps, la définition de la visualisation scientifique, de la Réalité Virtuelle ainsi que ses principales composantes. Également la différence entre la Réalité

Virtuelle et la visualisation de données scientifiques avec le positionnement de cette dernière par rapport à la composante interaction.

Étant donné qu'une application de visualisation scientifique interactive contient les trois parties fonctionnelles citées auparavant, la conception d'une telle application reste inaccessible pour les non spécialistes et les scientifiques. Pour la rendre accessible, il faut donc proposer une approche formelle afin de favoriser la séparation des préoccupations et raisonner rigoureusement pour démontrer la validité de ces applications. Il est nécessaire d'introduire un nouveau modèle de composants spécifique prenant en compte les caractéristiques de la visualisation scientifique interactive et de l'analyse visuelle (Visual Analytics). Le modèle présenté est basé sur de nouveaux moyens efficaces pour la construction et le déploiement d'applications dédiées à la visualisation scientifique interactive sur des architectures distribuées/parallèles.

ARCHITECTURE PAR COMPOSANTS

Walking on water and developing software from a specification are easy if both are frozen.

Edward V BERARD

SOMMAIRE

2.1	INTRODUCTION	19
2.2	HISTORIQUE DE LA PROGRAMMATION PAR COMPOSANTS	20
2.3	CONCEPTS DE LA PROGRAMMATION PAR COMPOSANTS	21
2.3.1	Composant logiciel	22
2.3.2	Interfaces	23
2.3.3	Connecteurs	24
2.3.4	Langage de description des architectures par composants	25
2.4	MODÈLES À COMPOSITION SPATIALE	25
2.4.1	Modèles généraux	25
2.4.2	Modèles spécifiques	30
2.4.3	Analyse & Synthèse	33
2.5	MODÈLES À COMPOSITION TEMPORELLE	35
2.5.1	Workflow	35
2.5.2	Systèmes de workflow scientifique	37
2.5.3	Analyse & Synthèse	45
2.6	MODÈLES À COMPOSITION SPATIO-TEMPORELLE	45
2.6.1	Systèmes de composant logiciel & workflow scientifique	45
2.6.2	Analyse & Synthèse	48
2.7	LA RECONFIGURATION DES ARCHITECTURES PAR COMPOSANTS	48
2.7.1	La reconfiguration centralisée	48
2.7.2	La reconfiguration auto-adaptative	50
2.7.3	Analyse & Synthèse	51
2.8	CONCLUSION	51

2.1 INTRODUCTION

Le but de ce chapitre est d'introduire les concepts fondamentaux, utilisés dans le cadre des architectures par composants. Généralement, la composition est un moyen de décrire la structure des applications et une telle structure reflète le raisonnement de la dimension de la programmation. Chaque modèle repose sur une dimension parmi

les deux dimensions orthogonales suivantes : l'espace et le temps. Le premier est dédié aux systèmes de composants logiciels et le deuxième est dédié pour les systèmes de workflow scientifique. L'intérêt de la composition spatiale est la définition de la structure des applications scientifiques indépendamment des aspects temporels explicites. La composition temporelle a comme intérêt la définition des relations entre les composants c-à-d elle exprime un ordre d'exécution des composants. Pour bénéficier des avantages des deux approches de composition, la composition spatio-temporelle combine, à tous les niveaux d'une structure d'application, les deux types de composition présentés.

Ce chapitre est divisé en six sections. La première introduit le cadre historique de l'architecture par composants. La deuxième présente les concepts liés à la composition spatiale avec des modèles exemples basés sur des composants. La troisième introduit, dans un premier temps, les concepts et les définitions utilisées permettant de construire une application basée sur un workflow. Et dans un deuxième temps, elle introduit des systèmes parmi les plus significatifs basés sur la composition temporelle. Dans la quatrième section nous présentons des modèles de composition spatio-temporelle. La sixième section présente quelques exemples représentatifs de reconfiguration des architectures par composants. Enfin nous concluons ce chapitre.

2.2 HISTORIQUE DE LA PROGRAMMATION PAR COMPOSANTS

La programmation par composants a émergé comme méthodologie de programmation pour des systèmes complexes et distribués. D'une manière générale, ce type d'architecture logicielle a été développé en se basant sur des composants représentant des entités autonomes qui interagissent avec leur environnement à travers des interfaces bien spécifiées et bien définies. Ces composants ne révèlent pas leur structure interne en particulier leur implémentation. Ce type de programmation favorise des caractéristiques comme la réutilisation, l'interchangeabilité, la séparation des préoccupations et la facilité de mise à jour.

Depuis les années 1990 et face à l'évolution rapide des exigences des systèmes et programmes informatiques, il était primordial de revoir et de réétudier les architectures logicielles existantes car elles ne répondaient pas aux besoins des chercheurs et des utilisateurs. Ces architectures logicielles décrivent d'une manière symbolique et schématique les différents éléments d'un ou de plusieurs systèmes, leurs intégrations et leurs interactions. Le terme d'architecture logicielle est utilisé depuis les années 1960 [156], mais il s'est vraiment imposé qu'à partir des années 1990 [128]. La figure 2.1 montre son évolution depuis 1960. Chronologiquement l'architecture basée sur la programmation structurée [210] est apparue en premier, ensuite l'architecture basée sur la décomposition fonctionnelle et puis celle basée sur le mode de communication client-serveur [72, 178]. Pendant la décennie 1980-1990, l'architecture 3-tiers, qui est une extension de l'architecture client-serveur [85], a vu le jour. Au fil des années, les

recherches académiques ont fait apparaître des architectures distribuées orientées objets [66, 157]. Enfin, la technologie des architectures logicielles à base de composants devient progressivement puissante et facilite le développement des systèmes [113]. Aujourd'hui, les approches par composants sont en pleine phase d'accroissement, elles sont reconnues comme étant des approches puissantes permettant d'améliorer significativement la manière de développer. Elles consistent à utiliser une approche modulaire de l'architecture d'une application. Cette dernière est construite en assemblant les composants compatibles entre eux.

Ces approches peuvent être classifiées en trois familles en fonction de la définition de la structure des applications et de l'ordonnancement des éléments construisant ces dernières :

- **composition spatiale** décrivant la connexion entre les composants lors de leur programmation en occupant des ressources matérielles qui peuvent être mémoires vives et virtuelles, réseaux, processeurs, etc. Ce type de composition permet d'exprimer l'architecture d'une application en se basant sur une représentation spatiale et en décrivant les composants qui doivent exister simultanément et peuvent communiquer indépendamment des contraintes temporelles explicites. La direction des communications est également orientée et elle est basée, dans la plupart des modèles, sur un utilisateur invoquant un service sur un fournisseur ;
- **composition temporelle** exprimant un raisonnement temporel avec une relation d'ordre entre les composants pour fournir une planification (control flow). La relation d'ordre est donnée par certaines structures de contrôle telles que des séquences, des branches ou des boucles essentiellement pour le passage de données d'un composant à l'autre ;
- **composition spatio-temporelle** intégrant les données qui traversent les composants (data flow) avec des fonctionnalités control flow pour une coordination précise. Elle permet de prendre en compte à la fois le couplage et l'optimisation de l'utilisation des ressources d'un système.

Plusieurs concepts sont utilisés dans ces approches. Ils sont présentés ainsi que leurs relations dans les sections suivantes.

2.3 CONCEPTS DE LA PROGRAMMATION PAR COMPOSANTS

Nous présentons ici le rôle et le fonctionnement des différents éléments d'une approche par composants.

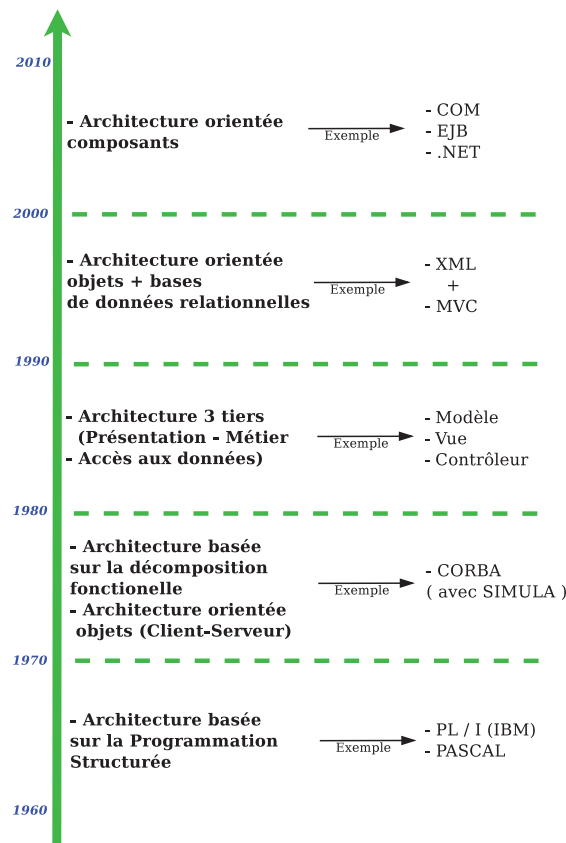


FIGURE 2.1 – L’histoire des architectures logicielles.

2.3.1 Composant logiciel

Un *composant logiciel* est un élément architectural qui encapsule la partie métier de l’application. Il existe plusieurs définitions dans la littérature d’un composant mais la plupart sont intuitives car elles se concentrent sur les aspects généraux d’un composant comme la définition donnée par Microsoft [68] :

« ...a piece of compiled software, which is offering a service ... »

D’autres détaillent la caractérisation d’un composant comme celle donnée par Sametinger dans [176] :

« Reusable software components are self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and a defined reuse status. »

par D’Souza et Wills dans [84] :

« A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger. »

par UML 1.3 specification (OMG 1999) dans [48] :

« *A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files.* »

et la définition la plus utilisée est celle de Szyperski dans [186] :

« *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party.* »

La majorité des définitions d'un composant se rejoignent sur le fait qu'un composant est une unité de calcul ou de stockage autonome, contenant du code applicatif ou métier afin de fournir un (des) service(s) bien déterminé(s) pouvant être appelé(s) depuis l'extérieur à l'aide d'*interfaces*. Il existe une séparation entre l'implémentation et l'appel d'un composant. Cette séparation permet de construire des applications par assemblage de composants qui peuvent être ainsi utilisés dans différents contextes. Cette définition d'un composant logiciel est dédiée aux modèles basés sur la composition spatiale et reste valable aussi pour les autres modèles basés sur la composition temporelle. Mais, un composant logiciel dans la composition temporelle est nommé une *tâche*. Cette dernière est une unité de travail logique dans un processus.

2.3.2 Interfaces

Une *interface* d'un composant est sa partie visible. Elle peut être définie comme son point d'accès c-à-d que cette dernière décrit les propriétés fonctionnelles ou les services du composant [43]. Un point d'accès est un élément d'architecture important qui fournit les descriptions et les protocoles des opérations. Ces interfaces sont des éléments d'application représentées par des *ports*. Chaque interface est associée à un composant unique c-à-d qu'une interface ne peut pas appartenir à plusieurs composants à la fois. Le rôle de ces interfaces est l'acheminement des données et la définition de données échangées entre les composants à savoir que ces interfaces sont typées. C'est à travers ces interfaces que les composants vont être connectés. On peut donc avoir des interfaces d'entrée ou des interfaces de sortie. Les interfaces d'entrée sont utilisées pour obtenir des données. Les interfaces de sortie au contraire fournissent les données résultantes produites par l'exécution d'un composant. Les interfaces d'un composant sont particulièrement importantes pour la composition et la personnalisation des composants par les utilisateurs. Elles lui permettent de trouver des composants adaptés et comprendre leurs usages, fonctionnalités, utilisations et restrictions. Une interface d'un composant comprend :

- une partie de la signature décrivant les opérations fournies par un composant ;
- une autre partie décrivant le comportement du composant.

2.3.3 Connecteurs

Certains modèles ont émergé pour donner une représentation explicite aux échanges des données entre les composants en proposant la notion de *connecteurs*. Un connecteur est un élément architectural au même niveau qu'un composant logiciel. Il modélise explicitement les interactions entre un ou plusieurs composants avec des règles qui les gouvernent [179]. Il agit comme médiateur et réalise une coordination entre les composants. Il garantit un certain protocole ou mode de communication. Un connecteur peut être implicite ou explicite. Une liaison réalisée par un connecteur implicite n'est qu'une liaison directe entre les interfaces des composants comme dans [142]. D'autre part, une liaison réalisée par un connecteur explicite permet d'offrir un assemblage de composants pour construire une telle application. Dans la plupart des travaux comme [149, 150], un connecteur comprend deux parties :

- la partie visible du connecteur qui correspond à son interface et qui permet la description des rôles des participants à une interaction ;
- la partie correspondante à la description de son implantation.

Selon [147] un connecteur peut être :

- connecteur direct : fournit une connexion directe simplement en formant des acheminements entre les ports. Par exemple, il transmet des appels de méthode à partir d'un port à l'autre, sans fournir de services d'interaction supplémentaires ;
- connecteur indirect : composition d'une ou plusieurs connexions en enrichissant les acheminements par une combinaison de données et de contrôle pour fournir des services d'interaction plus riches.

D'après ces deux catégories mentionnées ci-dessus, les connecteurs sont destinés à encapsuler les interactions ou les communications tandis que les composants sont destinés à encapsuler les calculs. En effet, ces deux catégories sont utilisées dans de nombreux modèles à base de composants avec deux type de variations [21, 82] :

- **connecteur endogène** : gère les interactions entre un port client d'un composant avec un autre port serveur du même ou d'un autre composant. Il peut être direct ou indirect, dans les deux cas, le connecteur ne contient pas une méthode interne. Ce connecteur intervient seulement pour transmettre une donnée vers un port client d'un composant ;
- **connecteur exogène** : cette variation détermine que le connecteur est responsable du contrôle (le composant est responsable de calculs), et est nécessairement un connecteur indirect. Il relie plusieurs ports et établit une séquence de méthodes dans ces ports. Ces méthodes sont internes au niveau du connecteur (contrairement aux connecteurs endogènes, où les invocations de méthode de port sont externes).

2.3.4 Langage de description des architectures par composants

Un *langage de description des architectures* (Architecture Description Language, ADL) est un langage formel ou semi-formel qui fournit des dispositifs pour modéliser l'architecture d'une application que l'on distingue de son implémentation. Ce langage offre une vision claire et détaillée de l'architecture d'une application concernant les dépendances existantes entre ses composants, leurs modes de communication et la politique de répartition des composants.

Il existe trois grandes catégories :

- langage académique comme ArchJava [15];
- langages de description d'architecture (ADLs) comme Wright [17];
- les standards comme AADL [89] et UML 2.0 [170].

Un ADL représente une modélisation des architectures comme le typage des interfaces, l'évolution et la configuration des composants et des connecteurs. Il peut être utilisé également pour décrire l'aspect structural des architectures des applications, par exemple, l'assemblage des composants et des connecteurs. Certains ADL peuvent offrir des informations supplémentaires comme le comportement d'un composant, les protocoles d'interactions et les propriétés fonctionnelles. De plus, ils peuvent être plus au moins extensibles comme dans l'ADL ACME [94] qui fournit la possibilité de définir des nouveaux types de composants, des connecteurs et des types de ports.

2.4 MODÈLES À COMPOSITION SPATIALE

Cette section est consacrée à la description de plusieurs modèles basés sur la composition spatiale, classés en deux catégories *modèles généraux* et *modèles spécifiques*. Chaque modèle sera présenté avec une description de ses différents éléments, de son langage de description d'architecture, de l'assemblage de ses éléments, du parallélisme de ses composants et des exemples de son utilisation.

2.4.1 Modèles généraux

2.4.1.1 CCM

Le CORBA Component Model (CCM) [6] a été développé pour fournir un modèle de composants distribués avec des composants hétérogènes. Cette hétérogénéité est gérée à l'aide d'une utilisation de plusieurs langages de programmation. De plus, le protocole standard IIOP (Internet Inter-ORB Protocol) assure l'interopérabilité des composants [103]. CCM supporte des applications distribuées orientées objet et développées selon le modèle client-serveur en se basant sur l'appel de méthodes distantes RPC. Ce modèle fournit une transparence pour le client quand il envoie des requêtes. Aussi, il offre une portabilité des applications afin qu'elles s'exécutent sur différentes architectures. Un composant CCM possède quatre types de ports de base, une fabrique et des attributs (voir figure 2.2) :

- Facette : interface fournie par un composant et qui sert de point de vue sur ce dernier c-à-d qui expose une fonctionnalité ;
- Réceptacle : interface de connexion et de déconnexion entre composants ;
- Source d'événement : point d'émission d'événements asynchrones ;
- Puits d'événement : point de réception de données asynchrones à partir d'un autre composant ;
- Fabrique(s) : home(s) est une nouvelle notion introduite dans CCM. Elle fournit des opérations pour gérer le cycle de vie des composants et les associations entre instances de composants ;
- Attribut(s) : principalement destiné(s) à être utilisé(s) pour la configuration des composants.

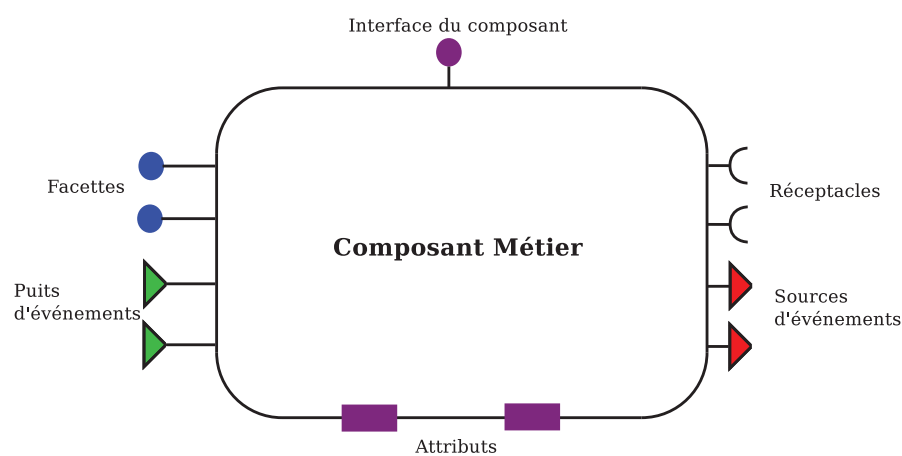


FIGURE 2.2 – Un composant CORBA et ses ports.

CORBA IDL (Interface Definition Language) décrit les interfaces qui supportent les fonctionnalités définies dans le corps d'un composant.

L'assemblage des composants CORBA [180] se réalise grâce à un descripteur XML en montrant comment les composants interagissent. Ce langage permet aussi de fournir une présentation détaillée de l'assemblage concernant les propriétés des composants, la génération de configurations par défaut pour les composants CCM et l'établissement des interconnexions nécessaires entre ces derniers.

Les composants du modèle CCM peuvent être utilisés pour concevoir des applications parallèles. Quelques travaux [130, 166] utilisent CCM dans les calculs scientifiques.

2.4.1.2 Le modèle FRACTAL

FRACTAL est un modèle qui appartient à un projet open source, pour la construction des systèmes à base de composants, développé par le consortium OW2¹. FRACTAL est un modèle de composants logiciels destiné à construire, déployer et gérer des

1. <http://fractal.ow2.org/>

systèmes complexes en utilisant différentes formes de composition et de connexion entre des composants, avec un langage de programmation indépendant. Les caractéristiques du modèle sont :

- hiérarchie des composants ;
- composants partagés ;
- capacités d'introspection ;
- reconfiguration.

Un composant FRACTAL [57, 69] est une entité d'exécution encapsulée supportant une ou plusieurs interfaces. Ces interfaces sont des points d'échanges (*port* dans d'autres modèles à base de composants) entre les composants exprimant leurs liaisons en termes d'interfaces client (représentant les services requis par le composant) et d'interfaces serveur (représentant les services fournis par le composant). Les interactions entre les composants s'achèment grâce à des connecteurs nommés *liaisons* qui sont des canaux de communication entre les interfaces. Un composant est soit un *composant client* demandant l'exécution d'un service à un autre composant par envoi de requête contenant le descriptif du service à exécuter et attendant la réponse de ce service par un message en retour, ou soit un *composant serveur* réalisant un service sur demande d'un client, et lui transmettant le résultat. Un composant comporte généralement deux éléments :

- une membrane qui fournit ou utilise deux types d'interfaces *internes* ou *externes* et qui contient un ensemble de *contrôleurs*. Les interfaces internes permettent de composer les sous-composants à partir de leurs interfaces externes afin de constituer le composant composite alors que les interfaces externes sont accessibles depuis l'extérieur du composant. La membrane fournit le contrôle du comportement d'un composant à l'aide de contrôleurs pour, par exemple, gérer le cycle de vie d'un composant c-à-d sa création, son démarrage et son arrêt ;
- un conteneur qui est un ensemble fini de sous-composants et de liaisons.

Les liaisons sont construites d'une manière explicite lors de la composition et peuvent représenter des voies de communication à distance entre les interfaces. Ceci permet la construction d'une configuration ou architecture répartie de composants FRACTAL. Il existe deux méthodes de liaison : *primitive* ou *composite*. La liaison primitive permet de relier une interface d'un composant client avec une interface d'un composant serveur. La liaison composite est constituée d'un ensemble de composants de liaison associés par des liaisons primitives. Le modèle FRACTAL possède un langage de description d'architecture FRACTAL ADL qui définit les configurations c-à-d l'assemblage et la composition des composants. FRACTAL ADL est basé sur XML et il est modulaire et extensible pour décrire les composants, les interfaces et les liaisons en particulier les constructions des membranes. En outre, ce langage peut décrire les informations concernant le déploiement, les comportements et toute autre préoccupation architecturale.

Le parallélisme sous FRACTAL est supporté pour que les utilisateurs puissent construire des applications parallèles. Ce modèle est utilisé pour concevoir les logiciels

distribués comme [109]. De plus, il est utilisé au niveau d'une méthodologie pour construire des modèles comportementaux de composants hiérarchiques, y compris les opérations de reconfiguration non structurelles [34].

2.4.1.3 Services Web

L'architecture orientée service (Service Oriented Architecture, SOA) [162] exploite le concept de *service* comme une unité principale pour développer et construire des applications complexes et distribuées. Cette architecture se base sur des technologies Web afin d'établir et gérer les communications entre ses composants. Dans ce contexte, ces composants sont appelés *services Web*. Un service Web est un module dont les interfaces décrivent les opérations et les structures de données utilisées pour mettre en œuvre sa fonctionnalité. Ainsi, ce module est exploité comme une boîte noire qu'on peut réutiliser et déployer. Un service Web peut être aussi utilisé comme programme de communication et d'échange de données entre applications hétérogènes non basées sur des services Web, ceci en se basant sur des technologies standards et des protocoles. Pour décrire leurs caractéristiques fonctionnelles et non fonctionnelles, les services Web se basent sur plusieurs standards comme WSDL (Web Services Description Language) [204], BPEL (Business Process Execution Language) [159] et WSCL (Web Services Conversation Language) [205]. Le protocole de communication qui permet de définir les formats des messages échangés entre les services est le SOAP (Simple Object Access Protocol) [206]. SOAP est un protocole RPC orienté objet basé sur XML. Il permet la transmission de messages entre les services et autorise l'invocation à distance des méthodes en mode requête-réponse. De plus, UDDI (Universal Description Discovery and Integration) [160] est une technologie utilisée pour enregistrer, publier et localiser les services Web. Cet annuaire est un intermédiaire entre les clients et les fournisseurs sur Internet.

La construction des applications autonomes et hétérogènes, en utilisant des services Web, se base sur la composition de ces derniers. La construction d'une composition peut se réaliser selon deux modes : *Orchestration* et *Chorégraphie* [165].

L'orchestration permet de décrire l'enchaînement, l'organisation et la coordination des services à l'aide du standard BPEL. Elle est centralisée par des définitions d'opérations explicites et par l'ordre d'invocation des services Web. Les services Web concernés sont sous le contrôle d'un processus central unique qui peut être un autre service Web qui est le coordinateur de l'orchestration (figure 2.3).

Ce processus coordonne l'exécution des différentes opérations de services Web participant au processus. Les services Web invoqués n'ont pas besoin de savoir qu'ils sont impliqués dans un processus de composition et qu'ils participent à la définition des processus métiers.

La chorégraphie ne dépend pas d'un coordinateur central (figure 2.4). Elle permet de concevoir une coordination décentralisée en utilisant le langage WS-CDL (Web Services Choreography Description Language) [207]. Ce type de composition est collaboratif où chaque service Web participant décrit son rôle dans son(ses) interaction(s),

en sachant exactement quand est ce qu'il va devenir actif et avec qui il va inter-opérer. Tous les services Web impliqués dans une chorégraphie doivent connaître les opérations à exécuter et leur déroulement. La chorégraphie est utilisée principalement pour échanger des messages dans les processus métiers publics.

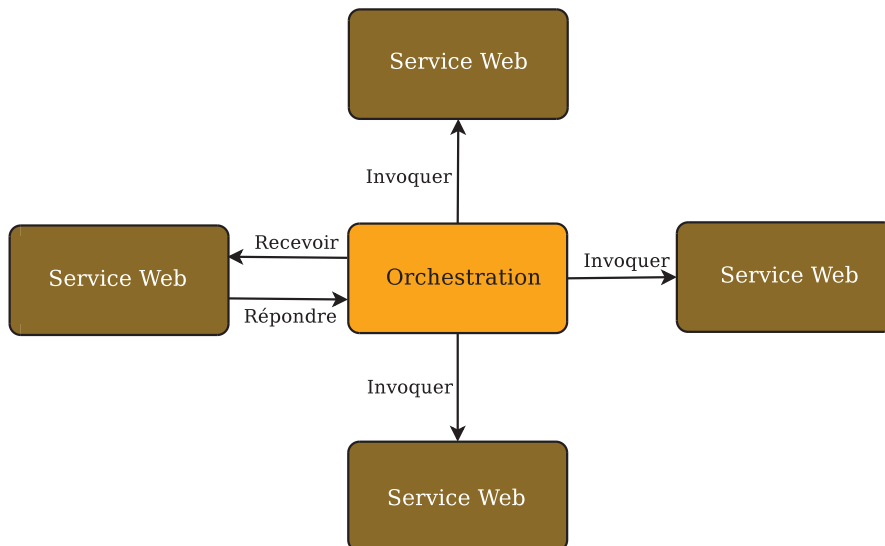


FIGURE 2.3 – L'orchestration des services Web.

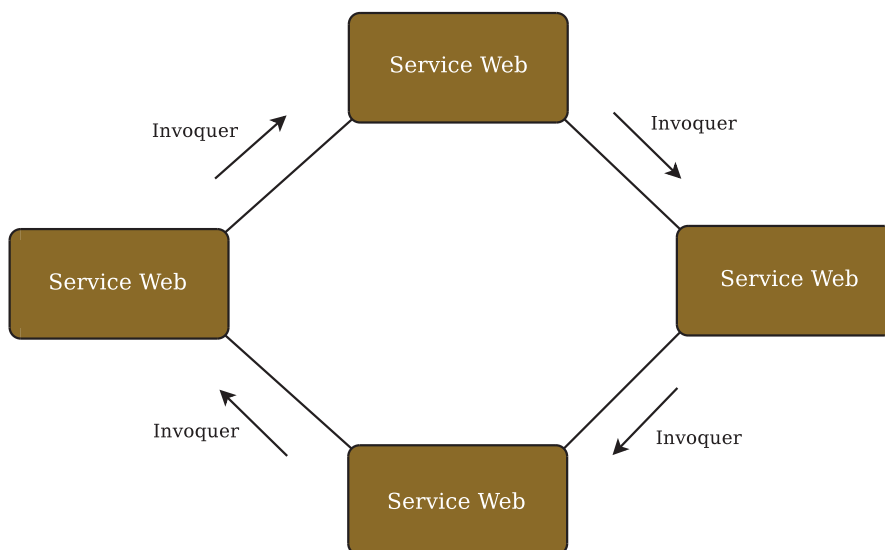


FIGURE 2.4 – La chorégraphie des services Web.

Dans l'orchestration, les services Web invoqués ne savent pas qu'ils appartiennent à un processus métier, par contre, dans la chorégraphie les services Web participants doivent savoir quand ils vont être actifs et avec qui ils vont interférer. De plus, la

chorégraphie pose plusieurs problèmes notamment la gestion d'erreurs et l'évolution dynamique.

Mais, elle permet aux services Web de s'administrer eux-mêmes, ce qui n'est pas le cas pour l'orchestration où les services Web sont dirigés par un seul maître. Le passage à l'échelle est difficile pour les processus complexes en utilisant l'orchestration, par contre, il est plus facile avec la chorégraphie.

Les architectures parallèles, traitant des données de manière simultanée, basées sur les services Web sont devenues dominantes. Il existe de nombreuses applications distribuées des services Web pour différents domaines, parmi ces applications nous citons [145, 188].

2.4.2 Modèles spécifiques

2.4.2.1 La norme IEC 61499

La norme IEC 61499 décrit un modèle de programmation des systèmes automatisés, proposée par International Electrotechnical Commission [202] et normalisée en Janvier 2005. Elle définit un modèle distribué pour diviser les différentes parties d'un processus d'automatisation industrielle et le contrôle de machines complexes en modules fonctionnels, appelés *blocs fonctionnels*. Chaque bloc fonctionnel représente une unité fonctionnelle de logiciel, qui est le plus petit élément d'un système de commande distribué. Il se compose de deux parties, la première est la partie ECC (Execution Control Chart) qui contient que des événements et des interfaces d'entrée et de sortie. Par contre, la deuxième partie est dédiée aux données et aux algorithmes à exécuter, qui se compose d'interfaces d'entrée et de sortie des données. Un bloc utilise et exploite une machine d'état ECC pour contrôler l'exécution de ses algorithmes (voir figure 2.5). Dans ce modèle, il n'existe pas de schémas de communication prédéfinis autrement dit il n'existe pas de connecteurs pour réaliser l'assemblage des blocs fonctionnels. Les événements et les données envoyés sont délivrés selon le principe *First-In-First-Out (FIFO)*.

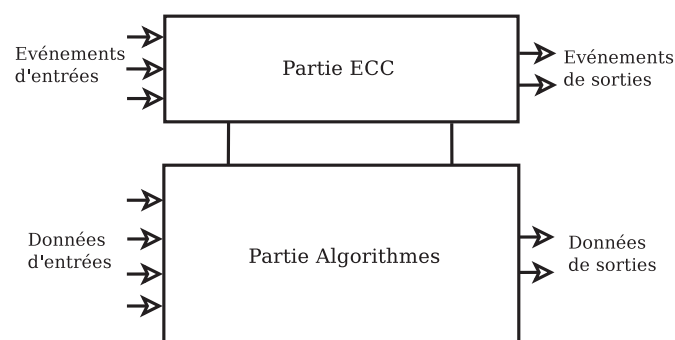


FIGURE 2.5 – Un bloc fonctionnel IEC 61499.

Chaque bloc fonctionnel a un comportement interne et est coordonné par un planificateur nommé *planificateur des ressources*. Ce planificateur permet de gérer la synchro-

nisation entre les événements et les données. D'abord, le bloc fonctionnel réceptionne les données et les événements, ensuite, le planificateur prévoit l'exécution de l'algorithme correspondant à l'événement reçu. Ainsi, l'algorithme concerné s'exécute et envoie un signal de sortie de données. Le planificateur est donc notifié pour que la partie ECC prenne le relais afin d'envoyer l'événement de sortie.

Un composant IEC 61499 est décrit avec le langage IEC 61131-3 [3]. Ce langage est utilisé par les environnements d'exécution comme ISaGRAF [2], FBDK [5] ou FORTE [4] pour construire et assembler les différents composants d'IEC 61499 afin de produire des applications embarquées non parallèles comme [45, 203].

2.4.2.2 Le modèle BIP

BIP (Behavior Interaction Priority) [36] est un modèle destiné au développement des systèmes embarqués en temps réel, en supportant la correction par construction et la vérification des applications construites. Ce modèle fournit une description et une composition des composants ainsi que des outils connexes pour analyser les modèles et la génération de code sur une plateforme dédiée. BIP construit ses composants à l'aide de la superposition de trois couches soient :

- Comportement (Behavior) : représente la couche inférieure spécifiant l'ensemble des transitions ;
- Interaction : est la couche intermédiaire qui contient l'ensemble des connecteurs décrivant les transitions qui représentent les comportements ;
- Priorité (Priority) : représente l'ensemble des règles de priorité décrivant les politiques d'ordonnancement des interactions.

Un composant BIP est constitué de :

- un ensemble de ports ;
- un ensemble de variable pour stocker les données locales ;
- un ensemble de transitions qui modélisent les étapes de calculs.

Pour assembler les composants, BIP utilise des connecteurs et des interactions. Un connecteur BIP est un ensemble de ports pouvant être impliqué dans une interaction. Chaque port contient un attribut *trigger* ou *synchron* pour modéliser les interactions. Les priorités sont utilisées pour filtrer les interactions qui sont réalisables. Une priorité est utilisée pour gérer les politiques d'ordonnancement et gérer, par exemple, l'absence de blocages.

BIP n'a pas un langage de description d'architecture mais il possède des outils pour réaliser l'assemblage des ses composants comme IF-toolset [54], Aldebaran [53] et DFinder [41]. Ces outils sont utilisés pour éditer la description d'un système BIP, pouvant être parallèle, l'analyser et générer le code C++ exécutable. BIP est utilisé dans plusieurs travaux comme [37, 181] et dans plusieurs projets comme ITEA/Spices² et OpenEMBeDD³

2. www.spices-itea.org/

3. www.openembedd.org/

2.4.2.3 Le modèle L2C

L2C (Low Level Component) est un modèle bas niveau proche de l'abstraction matérielle. Il permet de diminuer au minimum les pénalités de performances d'un modèle de composants primitif en C++.

Les composants de modèle L2C spécifient des points d'entrée et des points de sortie. Les points d'entrée sont utilisés pour créer et détruire une instance d'un composant c-à-d le configurer. Les points de sortie sont utilisés pour retourner et modifier les valeurs obtenues à partir d'autres composants. Les composants C++ de L2C supportent les deux types de ports de CORBA *uses* et *provides*.

L2C a un langage de description d'architecture basé sur l'XML et une API (Application Programming Interface). En effet, une application peut soit être décrite avec ce langage soit avec l'API, en connectant les composants.

La composition des objets L2C se réalise en utilisant le fichier de configuration LAD (L2C Assembly Descriptor). Ce fichier définit le nombre de composants à créer sur quels processeurs, leur connexion et leur configuration. Il existe deux types d'assemblage : C++/Fortran et CHARM++. L2C prend en charge la manipulation de l'ensemble de la structure de l'application à travers son assemblage [44] et les interactions entre composants pour les scénarios typiques de calcul de haute performance : tels que la mémoire du processus local partagé et l'invocation de fonction (C++ et Fortran), MPI et CORBA.

L2C peut être vu comme une extension de la compilation modulaire ou comme un modèle de composants distribués et/ou parallèles de bas niveau qui ne cache pas les problèmes du système. Il fournit une correspondance entre les langages C++, Fortran et CHARM++. Cette correspondance est basée sur des directives pré-processeurs et des exécutions courtes.

Ce modèle est utilisé pour concevoir les applications destinées au calcul haute performance dans le projet HLCM⁴.

2.4.2.4 Le modèle CCA

Le modèle CCA⁵ (Common Component Architecture) a été conçu pour le calcul haute performance, supporter le couplage de calculs parallèles et distribués et faciliter l'intégration de différents codes dans son environnement.

Le modèle CCA se compose des éléments suivants : *composants* qui sont des unités représentant des fonctionnalités. Ils sont utilisés comme boîtes noires avec des interfaces exposées afin de communiquer avec l'extérieur. Ces interfaces sont représentées de manière abstraite à l'aide de *ports* qu'un composant utilise pour interagir. Spécifiquement, le modèle CCA fournit des ports *provides* qui permettent aux composants de présenter des fonctionnalités et des ports *uses* pour faire appel à des composants sur un port provide à l'aide du protocole RPC.

4. <http://hlcm.gforge.inria.fr/>

5. <http://www.cca-forum.org>

L'*assemblage* des composants ne se réalise pas avec un langage ADL spécifique mais est géré par un *framework* qui assure la composition des composants et l'exécution des applications construites. Le framework connecte, par exemple, un port provide avec un port uses sans connaître les détails de l'implémentation des composants.

CCA exploite l'outil d'interopérabilité de haute performance Babel⁶ afin d'utiliser et mélanger plusieurs composants écrits en différents langages (C, C++, FORTRAN 77, Fortran 90/95, Fortran 2003/2008, Python, et Java) [26]. Babel fournit un environnement dans lequel tous les langages supportés sont traités. Cet outil supporte le calcul haute performance et le calcul distribué via les appels de méthodes distantes RMI (Remote Method Invocation) [129].

CCA a été conçu en mettant l'accent sur le calcul scientifique et ne force pas les scientifiques à utiliser un langage particulier de programmation. Les spécifications CCA sont écrites avec le langage SIDL (Scientific Interface Definition Language) [87], qui est semblable à CORBA IDL. Plus précisément, SIDL intègre les tableaux multidimensionnels dynamiques et intègre les nombres complexes. SIDL décrit les interfaces puis Babel passe à l'étape de compilation pour générer le *glue code*. Ce code sert uniquement à relier les différentes parties du code non compatibles et sert de proxy entre deux parties incompatibles de logiciels.

CCA permet de construire des applications contenant des composants s'exécutant en parallèle et il joue un rôle central au sein de plusieurs travaux tels que [25,55]

2.4.3 Analyse & Synthèse

Généralement, les modèles de composants présentés appartiennent à la composition spatiale car ils utilisent une composition basée sur une connexion spatiale des composants sans prendre en considération tout aspect temporel explicite.

Certains modèles de composants logiciels à savoir CCM, CCA, L2C et services Web se basent sur l'appel de méthode distante comme RPC pour CCM et CCA ou bien XML-RPC, JSON-RPC, SOAP et REST pour les services Web en mode requête-réponse. Cela facilite la mise au point et la portabilité des composants avec une coordination endogène mais au prix de l'indépendance et l'autonomie des entités des architectures distribuées. Par contre, FRACTAL, IEC 61499 et BIP reposent sur une communication exogène en se basant sur des connecteurs indépendants et autonomes, ce qui valorise l'intégration des applications hétérogènes dans des environnements distribués.

Chaque modèle a son propre langage d'assemblage qui permet de décrire, à l'aide d'une syntaxe définie, des assemblages de composants. Ce langage permet aussi d'explicitier la structure d'une application à construire afin de faciliter son développement. L'assemblage de composants définit les instances de composants d'une application et les connexions entre ces instances permettant une communication entre un port d'entrée (ou client) et un port de sortie (ou serveur). CCM a un langage d'assemblage spécifique à l'aide d'une syntaxe XML et FRACTAL possède son propre langage FRACTAL ADL définissant les différentes connexions entre ses composants. Le modèle IEC 61499

6. <http://www.llnl.gov/CASC/components/>

		Mode de communication	Langage de spécification	Langage d'assemblage	Parallélisme
Modèles généraux	CCM	Endogène avec RPC	CIDL	XML (format spécifique)	✓
	FRACTAL	Exogène	-	FRACTAL IDL	✓
	Services Web	Endogène Requête-Réponse	WSDL	BPEL/WS-CDL	✓
Modèles spécifiques	IEC 61499	Exogène	-	ECC	∅
	BIP	Exogène	Langage BIP	Langage BIP	✓
	L2C	Endogène avec MPI	-	LAD	✓
	CCA	Endogène avec RPC	SIDL	-	✓

TABLE 2.1 – Comparaison des modèles CCM, FRACTAL, services Web, IEC 61499, BIP, L2C et CCA.

exploite ECC pour définir l'architecture de l'application, ce qui implique une détermination des séquences d'opérations de ses blocs fonctionnels. Les modèles L2C, services Web et BIP supportent l'assemblage en utilisant LAD, BPEL/WS-CDL et langage BIP respectivement. Par contre, CCA n'utilise aucun langage pour décrire la structure de ses applications ce qui rend la compréhension des structures construites difficile.

FRACTAL, IEC 61499 et L2C n'ont pas de langage de spécification de l'architecture de l'application que l'on souhaite développer. Cette spécification représente une description formelle utilisée pour vérifier formellement que la réalisation finale respecte bien les attentes initiales. Elle existe pour CCM, CCA, service Web et BIP avec respectivement CIDL pour CCM, SIDL pour CCA, WSDL pour services Web et un langage spécifique dédié au modèle BIP.

La table 2.1 représente une comparaison entre les différents modèles représentés auparavant en se basant sur le mode de communication utilisé, le langage de spécification, le langage d'assemblage et est ce que le parallélisme est supporté par le modèle.

Après avoir présenté des modèles à base de composants logiciels utilisant la composition spatiale, dans ce qui suit, nous présentons des modèles de workflow basés sur la composition temporelle.

2.5 MODÈLES À COMPOSITION TEMPORELLE

Les approches à composition temporelle sont essentiellement des approches basées sur le *workflow* qui est une modélisation et une gestion de l'ensemble des tâches à accomplir et des différents acteurs impliqués dans la réalisation d'un processus métier.

2.5.1 Workflow

Le *workflow* a été standardisé par l'organisme WfMC (The Workflow Management Coalition) qui lui a donné une définition largement acceptée [118] :

« Workflow is the computerized facilitation or automation of a business process, in whole or part... »

Les travaux de recherche existants sur les workflows couvrent plusieurs domaines à savoir la météorologie, la bio-informatique, la physique et la chimie [102, 115]. Grâce à cet élargissement de l'utilisation des workflows, les chercheurs peuvent facilement intégrer et accéder à divers outils ou données distribuées pour développer leurs propres protocoles de recherche afin de réaliser des analyses scientifiques. Ces processus sont réalisés à l'aide de la notion *workflow scientifique*. Ce dernier combine les données et les processus en un environnement configurable et structuré mettant en œuvre des solutions semi-automatiques de calcul pour un problème scientifique. Souvent les résultats des calculs sont envoyés vers des interfaces graphiques en combinant différentes technologies.

Un workflow scientifique permet de réaliser les trois tâches basiques suivantes : (1) gérer les procédures, (2) coordonner un ensemble de tâches c-à-d les charges et les ressources et (3) superviser le déroulement et l'exécution des opérations [118]. Un workflow scientifique peut se reformuler sous forme de réseau avec des nœuds représentant des tâches c-à-d des activités et les flux représentés par des transitions.

Un workflow scientifique est une description abstraite d'étapes requises pour exécuter un processus particulier et un flux d'informations, un exemple est représenté par la figure 2.6. Chaque étape est définie par un ensemble d'activités qui doivent être réalisées. Dans un workflow scientifique, un travail passe à travers différentes étapes dans l'ordre indiqué, du début jusqu'à la fin, et les activités de chaque étape sont exécutées soit manuellement soit automatiquement.

Les workflows scientifiques peuvent se définir, se gérer et s'exécuter par les systèmes de gestion de workflow scientifique (workflow management system). Ces systèmes fournissent un environnement de configuration et de surveillance des workflows scientifiques donnés afin de garantir leur coordination. La WfMC [118] a fixé le rôle de ces systèmes par cette définition :

« A Workflow Management System consists of software components to store and interpret process definitions, create and manage workflow instances as they are executed, and control their interaction with workflow participants and applications... »

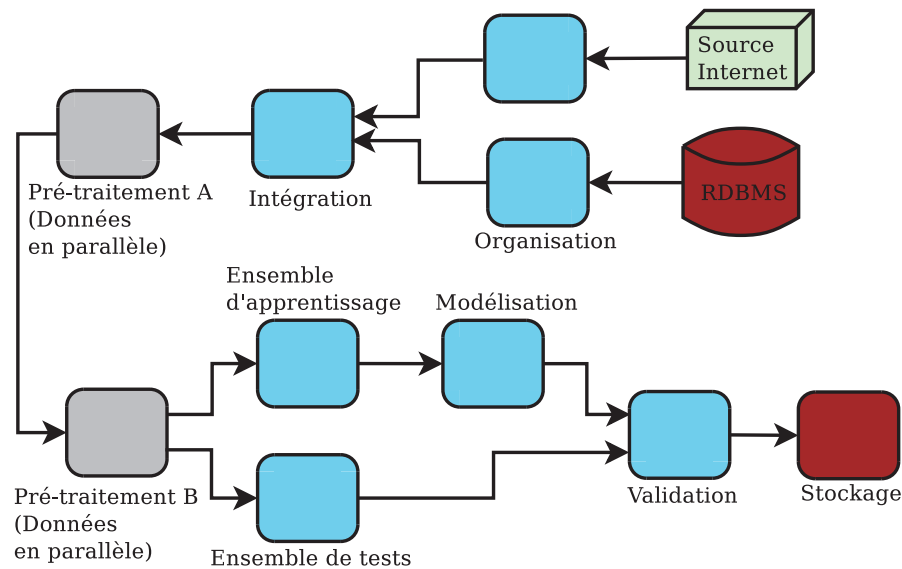


FIGURE 2.6 – Exemple d'un workflow scientifique.

Van Der Aalst [200] montre comment les systèmes de gestion de workflow scientifique se sont développés :

- 1965 - 1975 : **décomposition d'applications**. Les applications tournent directement sur des systèmes d'exploitation et ne possèdent pas d'interface utilisateur. Il n'y avait pas d'échanges de données entre les différentes applications. Ainsi, il fut impossible de combiner différentes sources de données ;
- 1975 - 1985 : **gestion des bases de données**. Après la naissance des systèmes de gestion de base de données (database management system), la combinaison des données gérées par plusieurs applications, la définition de la structure des données, le stockage de données pouvant se réaliser en une seule fois à l'aide d'une base de données ;
- 1985 - 1995 : **gestion de l'interface utilisateur**. Durant cette période, le développement des applications interactives a connu une croissance grâce à la réalisation des interfaces dédiées aux utilisateurs. Ces interfaces sont des dispositifs permettant à un utilisateur de manipuler son application ;
- 1995 - 2005 : **gestion de workflow**. Il est devenu attrayant d'isoler les procédures et le traitement de ces derniers afin de trouver une solution pour les séparer. Cette isolation offre des avantages intéressants car la maintenance, le changement ou la modification d'une procédure deviennent plus faciles à gérer.

Durant ces années, l'idée principale de ce changement se base sur la séparation des applications. Cette séparation offre une visibilité et une compréhension plus claires aux applications qui peuvent être plus facilement maintenues. Une application est un enchaînement ordonné de composants répondant à un certain schéma et aboutissant à un résultat déterminé. Cet enchaînement est obtenu grâce à une composition, c-à-d l'assemblage des composants, en utilisant des patrons de composition.

2.5.1.1 Composition des composants

La composition des composants est l'acte de regrouper les données élémentaires des étapes de traitement dans les activités de workflow [198]. Elle définit l'ordre d'exécution des composants à l'aide de flux de contrôle qui s'appuient sur des structures séquentielles permettant d'exécuter des composants, l'un après l'autre, ou sur des structures conditionnelles qui permettent de tester la satisfaction des conditions sur les valeurs des données. Ces données traitées par chaque composant sont spécifiées à l'aide de flux de données.

Nous allons présenter et analyser des exemples de grands systèmes de workflow scientifique qui intègrent l'automatisation de l'exécution de différents composants (tâches) en prenant en compte explicitement les dépendances temporelles. Traditionnellement, ces systèmes sont répartis en deux grandes familles, l'une pour l'orchestration de contrôle des processus et l'autre pour le calcul des données [70], selon trois aspects :

- **workflow** : peut prendre un certain nombre de formes, telle qu'une série d'unités fonctionnelles ou un ensemble des composants. Elle expose aussi les dépendances entre ces éléments en définissant l'ordre dans lequel les unités doivent être exécutées ;
- **data flow** : ils sont conçus pour supporter des applications pilotées et dirigées par les données. Le Data flow qui circulent entre les activités du workflow expriment les dépendances entre les différents composants des applications ;
- **control flow** : dans un workflow dirigé par le contrôle, les connexions entre les tâches (composants) dans un flux représentent un transfert (passage) du contrôle de l'activité précédente vers la suivante. Ce mécanisme s'effectue à l'aide de structures de contrôle telles que les conditions, les itérations, les boucles ou les séquences.

2.5.2 Systèmes de workflow scientifique

2.5.2.1 Discovery Net

Le système *Discovery Net* a été conçu principalement pour soutenir l'analyse des données scientifiques fondées sur des services Web distribués [18, 175]. Il a été mis en œuvre par l'Imperial College de Londres et a été développé dans le cadre du projet Discovery Net pilote (2001-2005). Les objectifs de Discovery Net sont : d'étudier et de résoudre les principaux problèmes dans le développement d'une plateforme e-science qui importe des données générées par une grande variété de capteurs à haut débit. Il est utilisé dans plusieurs domaines comme les sciences de la vie [98], la surveillance géo-risques [174], la modélisation de l'environnement et les énergies renouvelables [108].

Il est basé sur une architecture multi-niveaux, avec un serveur de workflow fournissant un certain nombre de fonctions nécessaires pour la création et l'exécution d'un

flux tels que l'intégration et l'accès aux outils de visualisation. Discovery Net se compose de plusieurs services qui peuvent être exécutés à partir de sites distants et qui peuvent être considérés comme des boîtes noires avec des interfaces d'entrée et de sortie bien connues. Ces services sont reliés entre eux sous forme de graphes acycliques en définissant des séquences d'opérations.

Workflow Les workflows dans Discovery Net sont représentés et stockés à l'aide du DPML (Discovery Process Markup Language) [185] qui est un langage de représentation de workflow basé sur XML en supportant à la fois le workflow d'analyse et l'orchestration de plusieurs workflows. Dans DPML, chaque processus est réutilisable et peut être encapsulé et partagé comme un nouveau composant (service) sur des grilles pour d'autres utilisations. Chaque composant contient un nombre donné de paramètres, de ports d'entrée et de ports de sorties, figure 2.7. Les services sont connectés entre eux via des ports et ces connexions sont représentées par des arcs dans le graphe dont les nœuds sont les composants exécutables.

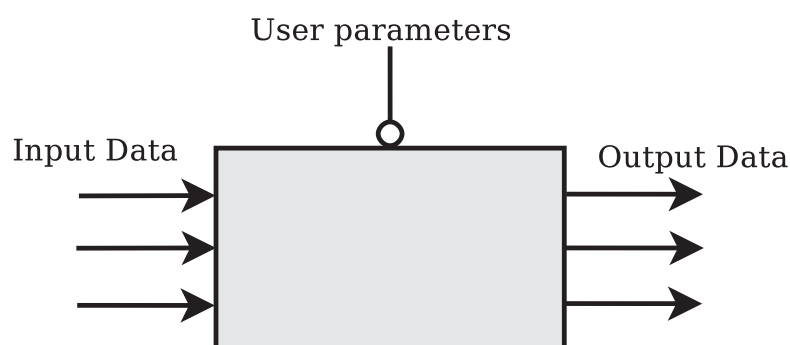


FIGURE 2.7 – Exemple d'un composant dans Discovery Net .

Discovery Net réalise une séparation entre le flux de données et le flux de contrôle au sein d'un flux de travail scientifique dans le but de mieux contrôler et coordonner le flux de données.

Data flow Discovery Net représente son workflow comme graphe de dépendance acyclique. Dans un graphe, l'exécution d'un des nœuds de l'extrémité permet de déclencher l'exécution de tous les nœuds précédents. Les nœuds sont décrits par des méta-données en définissant les types des données. Le type par défaut des données dans ce système est une table relationnelle. Donc au niveau de Discovery Net, le workflow est typé ce qui permet de réaliser et assurer une classification des composants par rapport à leurs entrées et sorties.

Control flow Le flux de contrôle se base sur des composants particuliers comme les composants de synchronisation, les composants conditionnels et les composants

boucles. Le rôle de ces composants de flux de contrôle est la réalisation de l'orchestration des flux de données, en précisant l'ordre de chaque flux de données. Parmi les opérations de flux de contrôle nous trouvons *Test*, *WaitForAll*, *While*, etc.

2.5.2.2 Taverna

Taverna est un outil open source pour la conception et l'exécution des workflows basés sur les services Web pour les bio-informaticiens et il est destiné à soutenir l'automatisation des processus complexes. Il est créé dans le cadre du projet myGrid et financé par l'OMII Royaume-Uni [184, 213]. Dans Taverna, un flux de travail est considéré comme un graphe de composants appelés processeurs, chacun d'entre eux transforme un ensemble de données d'entrée en un ensemble de données de sortie [70]. Ces flux sont représentés à l'aide du langage SCUFL (Simple Conceptual Unified Flow Language) en utilisant la syntaxe XML [161]. Taverna permet aux utilisateurs d'intégrer d'autres composants logiciels tels que ceux basés sur SOAP/WSDL, des services Web REST [151, 169] et il peut aussi invoquer des composants basés sur *R* [137].

Workflow Le flux de Taverna est représenté par le langage SCUFL. Il exploite le web pour construire son flux en prenant les URIs de la description WSDL des interfaces. Dans SCUFL un workflow est un réseau de composants et de liens et est représenté sous forme d'un DAG (Directed Acyclic Graphs), la figure 2.8 montre un exemple de ce type de graphe. Un workflow sous SCUFL contient plusieurs paramètres d'entrée nommés *sources*. Ces sources ont un nom unique dans un document SCUFL, de même les sorties d'un workflow sont nommées des *puits*. Les puits sont associés à des métadonnées (avec le type MIME⁷). Dans un graphe de workflow sous Taverna, il existe deux sortes de liens. Les liens de données qui relient les ports pour acheminer des données, les ports étant typés et les liens de coordination qui contiennent des contraintes de dépendance pour contrôler le flux d'exécution.

Data flow Dans Taverna le modèle de flux de données se compose de flux d'entrée et de flux de sortie [191]. Le flux d'entrée est décrit comme un graphe avec des nœuds composants qui peuvent l'exécuter sur l'entrée prévue, ce flux est représenté par des arcs entrants. Le flux de sortie est transmis à d'autres composants et est représenté par des arcs sortants. Dans ce modèle, l'ordre dans lequel les composants sont exécutés est principalement déterminé par l'ordre dans lequel les données apparaissent sur les différentes entrées. Dans le cadre du traitement normal, un port d'entrée peut recevoir une valeur d'un type qui ne correspond pas exactement à son type déclaré, par exemple, un composant *A* qui produit une valeur de type *list of strings* peut légalement être connecté à un composant *B* avec un port de type d'entrée *string*. Le résultat de cette connexion est représenté sous SCUFL par une fonction $f_B(f_A)$. Un port de sortie peut être connecté à plusieurs ports d'entrée. Dans ce cas, les éléments de données

7. Actuellement appelé *Internet media type* et c'est un identifiant de format de données sur internet en deux parties : un type et un sous-type et d'un ou plusieurs autres champs au besoin.

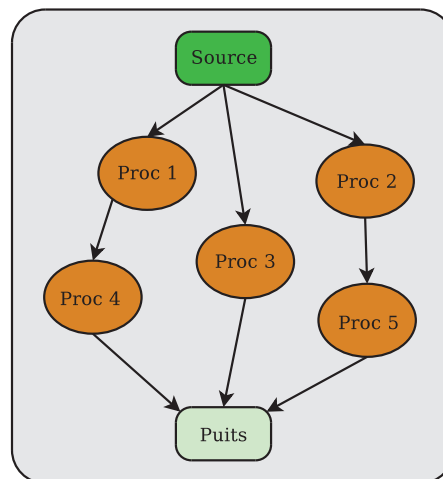


FIGURE 2.8 – Exemple d'un DAG avec des nœuds de données et des nœuds de contrôle.

sont diffusés sur tous les ports d'entrée connectés. De même, plusieurs ports de sortie peuvent être reliés à un port d'entrée unique. Dans ce cas, les éléments de données sont mis en tampon dans le port d'entrée en fonction de leur ordre d'arrivée.

Control flow Les liens de coordination et la structure conditionnelle sont les seules structures de contrôle disponibles dans Taverna. La structure *loop*, par exemple *while* et *for*, n'est pas présente, cependant une forme limitée d'itération est disponible en enveloppant des objets de type *date* dans la liste et en spécifiant la stratégie d'itération. Mis à part les liens de coordination, le workflow est complètement lié à la présence ou à l'absence de données dans les ports d'entrée d'un processeur : un processeur se déclenche si et seulement si l'ensemble de ses ports d'entrée contiennent des données adéquates.

2.5.2.3 Triana

Triana [144, 192] a été initialement développé comme un environnement de résolution des problèmes basés sur le flux de travail visuel. Il est développé à Cardiff University sous le projet GEO600 [31] et est utilisé comme outil open source d'analyse rapide de données d'onde. Il propose une interface graphique pour la conception de flux de travail avec des outils d'analyse de données. Le workflow sous Triana a été d'abord construit à partir des outils Java et exécuté sur des machines locales ou à distance à l'aide de Java RMI (Remote method invocation). Il a connu une grande évolution et peut supporter le calcul distribué.

Workflow Triana peut connecter plusieurs outils hétérogènes par exemple les services Web, Java, unités et services JXTA⁸, dans un workflow. Il utilise son propre langage de workflow personnalisé, basé sur une syntaxe XML simple, pour la construction des applications. Chaque représentation a une structure non-DAG et définit la liste des composants participants, les connexions entre elles et les composants hiérarchiques.

Triana peut utiliser d'autres représentations externes telle que le langage BPEL. Ce dernier est, en particulier, utilisé dans les architectures distribuées avec la création d'interfaces graphiques intuitives et des mécanismes correspondants pour distribuer ses composants sur le P2P (peer-to-peer) et les environnements de grilles de calcul.

Un composant fonctionnel sous Triana est appelé une *unité* (unit) avec une interface définie (port). Les unités sont connectées entre elles grâce à des connexions qui se nomment *câbles* (cables, en anglais) pour construire des workflows. Une unité dispose de plusieurs propriétés comme un identifiant, des ports d'entrée, des ports de sortie, un certain nombre de paramètres facultatifs, etc. La spécification d'une unité est codée en XML avec un format similaire à WSDL (Web Services Description Language) [13].

Data flow Un flux de données sous Triana se déplace à partir d'une unité source vers une unité destination. Ce mécanisme n'est pas le seul supporté par cet outil, mais, d'autres mécanismes comme la fusion, sont aussi utilisés. À l'aide des mécanismes existants sous Triana, il est possible de réaliser des exécutions en parallèle sur plusieurs ensembles de données. Une composition supporte la réalisation de la hiérarchie des unités sous forme d'un regroupement d'unités connectées à l'intérieur d'un niveau supérieur à l'aide d'une seule unité.

Control flow Les dépendances Triana entre les composants sont principalement des dépendances de données. Cependant, Triana prend également en charge la représentation des dépendances de flux de contrôle grâce à des messages spéciaux contrôlant les déclenchements entre les composants.

Les deux structures utilisées sous Triana sont : *If* et *Switch*. Avant n'importe quelle exécution, un composant attend initialement des données sur un port d'entrée (étape pre-loop). Ensuite, il itère l'exécution d'une tâche ou une séquence de tâches. Après la réception d'une donnée sur le port d'entrée, la condition de sortie définie est évaluée. Si cette condition est satisfaite, la valeur de donnée de sortie est envoyée vers le port de sortie. Si une donnée reçue sur un port d'entrée donné ne satisfait pas la condition de sortie, alors elle va être ré-envoyée vers son port d'entrée. En général, ce comportement expliqué d'un composant Triana est basé sur une boucle (Loop), une condition (If) et la sélection (Switch). Ce composant se nomme *composant loop*.

8. JXTA est un projet Open Source lancé par Sun Microsystems. Les protocoles JXTA sont définis comme un ensemble de messages XML qui permettent n'importe quel appareil connecté à un réseau d'échanger des messages et de collaborer indépendamment de la topologie du réseau sous-jacent.

2.5.2.4 Pegasus

Le système *Pegasus* [77,78] (Predicated Explicit GAted Simple Uniform Ssa) a été développé à l'Université du Sud de la Californie depuis 2001. C'est un système qui permet d'exécuter, gérer et déboguer des workflows complexes sur une large variété d'environnements : machine locale (Local desktop), Condor local [193], Cluster campus local, Grid ou bien des nuages commerciaux ou académiques. Pegasus est utilisé par un certain nombre d'applications dans des domaines variés comme l'astronomie, la bio-informatique, la science de tremblement de terre, la physique des ondes gravitationnelles, sciences de la mer, la limnologie et d'autres.

Le système de gestion de workflow Pegasus peut gérer l'exécution d'une application formalisée comme un workflow en mettant en correspondance sur des ressources disponibles des composants du workflow à exécuter dans l'ordre de leurs dépendances [189].

Workflow Les workflows abstraits conçus par un scientifique sont indépendants des ressources où ils seront exécutés. Cela permet à un scientifique de se concentrer sur la conception de workflow plutôt que sur la décision des ressources physiques à utiliser. Pour cela, Pegasus cherche à trouver une cartographie des composants à exécuter avec les ressources disponibles en utilisant des techniques de planification de l'Intelligence Artificielle. Le workflow abstrait sous Pegasus est défini comme un graphe acyclique orienté (DAG), composé des composants et des dépendances de données. Les composants échangent les données entre eux sous forme de fichiers.

La description de workflow dans ce système est réalisée à l'aide du langage haut niveau DAX (Directed Acyclic graph in XML). Ce langage fournit une description d'un workflow abstrait au format XML qui est utilisé comme entrée principale dans Pegasus. DAX est un langage dépourvu de description de ressources et des emplacements de données. Il se réfère aux codes de transformations logiques et à des données sous forme de fichiers logiques.

Chaque description DAX contient tous les composants qui vont effectuer le calcul, leur ordre d'exécution et pour chacun d'entre eux les entrées nécessaires, les résultats attendus, et les arguments avec lesquels le composant doit être invoqué. Le workflow Pegasus peut être défini de manière hiérarchique où le nœud d'un DAX représente un autre nœud DAX. La possibilité de définir des workflows permet à l'utilisateur de construire des structures plus complexes.

Data flow Les nœuds du graphe de la représentation sous Pegasus représentent des *opérations*, les arêtes contiennent la valeur du flux et contrôlent les dépendances entre les composants. Un nœud peut envoyer une valeur produite vers plusieurs autres nœuds, cela est représenté par plusieurs arêtes sortantes avec une arête pour chaque envoi. Les arêtes entrantes d'un nœud se nomment *Inputs* et les arêtes sortantes s'appellent *Outputs*. Les données sont produites par la source d'une arête, elles sont trans-

portées par l'arête et elles vont être consommées par la destination. En général, tous les nœuds ont besoin de toutes les données d'entrée pour calculer un résultat.

Control flow Le workflow est envoyé à un système de gestion et de contrôle de workflow qui est hébergé dans une autre machine appelée l'hôte de soumission (submit host). Cette machine peut être l'ordinateur portable d'un utilisateur ou un serveur. Dans le modèle de Pegasus, c'est le système de gestion de workflow qui est responsable de la traduction des composants, c-à-d la traduction d'une description de workflow abstrait vers une description de workflow exécutable du calcul, de l'exécution de ces composants, de la gestion des données (c-à-d le repérage, l'organisation, l'interprétation des fichiers échangés entre les composants dans le workflow), du suivi des exécutions et de la gestion des échecs. Ce système de gestion se compose de cinq principaux composants particuliers [78] : mapper Pegasus, moteur d'exécution local (Local execution engine), planificateur d'emploi (Job scheduler), moteur d'exécution à distance (Remote execution engine) et le composant du suivi (Monitoring component).

2.5.2.5 Kepler

Kepler [19] est un moteur open-source de construction, composition et orchestration des workflows scientifiques. Il est construit en se basant sur le système *Ptolemy II* [58]. Il est développé par une équipe basée à l'Université de Californie à Berkeley et est utilisé pour concevoir et exécuter différents workflows dans plusieurs domaines comme la biologie, l'écologie, la géologie, la chimie, l'astrophysique, etc.

Kepler utilise un *directeur* qui contrôle les séquencements et les communications d'un workflow. La conception dans Kepler est centrée sur des composants nommés *acteurs*. Ces derniers sont des blocs de calcul réutilisables et indépendants, par exemple, le modèle d'exécution, ou le modèle de calcul (Model of Computation, MoC) tels que : les services Web, les appels de base de données, transformateurs de données, ou des étapes analytiques, etc. Les acteurs représentent des opérations ou des sources de données avec un ensemble de ports d'entrée, de sortie ou mixtes qui agissent comme des interfaces d'extrémité pour des connexions acheminant des données. Ils consomment des données à partir d'un ensemble de ports d'entrée et écrivent ou produisent des données dans un ensemble de ports de sortie.

Workflow Les utilisateurs scientifiques peuvent utiliser Kepler facilement en échangeant, archivant et exécutant les workflows sous un format simple basé sur XML grâce à l'utilisation du langage MoML (Modeling Markup Language) [135]. Ce langage est utilisé pour préciser les interconnexions des composants hiérarchiques et paramétrés. Le langage MoML décrit l'organisation des données comme les attributs *nom* et *classe*, la définition des entités avec toutes leurs propriétés telles que les ports d'entrée, les ports de sortie, les relations et les liens.

Data flow Les données sous Kepler sont définies sous forme de jetons. L'exécution de chaque nœud, c-à-d composant, se réalise à l'aide de la consommation de jetons qui sont en entrée, ce qui implique la production des résultats sous forme de jetons en sortie. Par défaut, les données sous les acteurs Kepler se traitent localement comme un thread Java (local Java threads). Mais le système permet également de réaliser des exécutions distribuées à l'aide des services Web et des services Grid. Les acteurs dédiés aux services Web et les services Grid permettent aux utilisateurs d'utiliser les ressources de calcul distribuées pour un seul workflow. Ces acteurs de services Web fournissent à l'utilisateur de Kepler un moyen ou une interface pour exécuter facilement n'importe quel service Web défini par WSDL.

Control flow Kepler fournit une caractéristique intéressante qui est la séparation de data flow et du control flow représentant une coordination de workflow. Cette séparation est réalisée par plusieurs modèles de calcul que Kepler hérite du système Ptolemy et est représentée par plusieurs directeurs ou composants particuliers de même niveau qui peuvent être utilisés séparément ou conjointement pour contrôler le flux, par exemple :

- Data flow Synchrone (Synchronous Data Flow) [134] : qui est un réseau de flux de données synchrone nommé réseau de *Kahn* [124]. Au niveau de chaque activation un composant produit et consomme, respectivement, un nombre fixe de jetons de données sur chacun de ses canaux entrants et sortants. Pour activer chaque composant, il faut avoir le nombre de jetons nécessaires sur ses canaux d'entrée pour les consommer. Ce type de réseau est adapté pour la modélisation des autres data flow comme les calculs mathématiques ou la manipulation de tableau ;
- Réseau de Processus (Process Network) : ce directeur, contrairement au directeur Data flow Synchrone, ne calcule pas statiquement le temps d'activation. Mais dans un workflow du réseau de composants, chaque composant a un thread Java indépendant et le workflow s'active avec la disponibilité des données. Le directeur du Réseau de Processus autorise l'exécution de composants en parallèle ;
- Data flow Dynamique (Dynamic Data Flow) : il exécute un workflow dans un thread d'exécution unique, ce qui signifie que les tâches ne peuvent être effectuées en parallèle. Ce directeur ne fait aucune tentative pour prévoir une exécution du workflow. Il est exploité juste pour les flux utilisant des structures de type *Boolean*, précisément les structures *BooleanSwitch* et *if-then-else* ;
- Événement Discrets (Discrete Event, en anglais) : c'est un directeur qui joue le rôle de superviseur des workflows quand les événements se produisent à des instants discrets tout au long d'une durée. Donc, il est bien adapté et destiné à la modélisation de systèmes orientés-temps tels que les systèmes de files d'attente, les réseaux de communication, les temps d'attente, etc.

2.5.3 Analyse & Synthèse

Les modèles de workflow présentés ci-dessus sont basés sur une composition dans le temps car elle évoque un raisonnement ou une représentation temporelle explicite.

Dans ces systèmes à composition temporelle, l'expression du workflow, du data flow et du control flow diffère d'un système à l'autre. Tous ces systèmes sont data flow et control flow. Discovery Net est principalement un système de data flow, avec un modèle de contrôle ajouté comme une couche de coordination de haut niveau. Par contre, Taverna, Triana, Pegasus et Kepler sont des systèmes de data flow et control flow.

Dans Discovery Net, Taverna et Pegasus le flux est représenté sous forme de graphe acyclique DAG avec un data flow typé pour Discovery Net et Taverna et un flux décrit par le langage de haut niveau DAX dans Pegasus. La structure du workflow de Triana et Kepler est sous forme de graphe non-DAG, avec une représentation XML des données dans Triana et une représentation WSDL pour Kepler.

En outre, le control flow s'exprime d'une façon variée soit explicitement ou implicitement. Dans Discovery Net et Pegasus, le contrôle est intégré comme des modules spécifiques. De même, Taverna contient des structures conditionnelles mais sans la structure *Loop*. Alors que dans Triana, le control flow est basé sur une dépendance entre les données grâce aux deux instructions *If* et *Switch*. Dans Kepler, le control flow est assuré par des composants nommés des directeurs ce qui le rend explicite. La séparation entre le control flow et le data flow aide les scientifiques à continuer de développer et améliorer facilement les systèmes de workflow scientifique en se basant sur leurs besoins. Cela rend ces systèmes plus expressifs au niveau de leur intégration et de leur exécution.

2.6 MODÈLES À COMPOSITION SPATIO-TEMPORELLE

Les modèles à composition spatio-temporelle sont représentés par trois systèmes exemples de calculs scientifiques : ICENI, STCM et FlowVR. Ces environnements sont établis en utilisant une composition spatiale et temporelle permettant de bénéficier pleinement des apports couplés des deux types de composition.

2.6.1 Systèmes de composant logiciel & workflow scientifique

2.6.1.1 ICENI

ICENI (Imperial College e-Science Networked Infrastructure) [92] est un système à base de composants pour les grilles de calcul qui prend en charge les activités e-science. Son rôle est de fournir des applications avec l'accès aux ressources matérielles et logicielles sous-jacentes d'une manière intelligente, en masquant la nature hétérogène des ressources. L'objectif d'ICENI est l'ordonnancement efficace des composants

d'une application de manière à optimiser leur placement sur des ressources d'exécution. Un composant ICENI possède un nom, des ports d'entrée, des ports de sortie et des propriétés.

La composition sous ICENI se réalise soit textuellement soit graphiquement. Une application se construit à partir de composants connectés par leurs ports sous la forme d'un graphe non-DAG. ICENI se compose d'un certain nombre d'outils et de systèmes qui utilisent le langage CXML (Component eXtensible Markup Language) qui est basé sur le langage XML. Ce langage est utilisé pour décrire les méta-données des composants abstraits, les implémentations de composants, des ressources et applications. Chaque méta-donnée d'un composant contient une description sous forme de workflow de son comportement interne décrit par les développeurs. En effet, chaque modification d'une application a besoin d'une connaissance approfondie des comportements des composants avant leur déploiement et leur implémentation.

ICENI calcule un plan de déploiement spatial optimisé pour le comportement global d'une application construite à partir des différentes connexions entre les composants. L'optimisation spatiale d'ICENI est relative car les workflows sont générés à partir des descriptions des comportements des composants. Ce raisonnement spatial prend en compte un raisonnement temporel qui est déduit par ICENI et non par l'utilisateur.

2.6.1.2 STCM

STCM (Spatio-Temporal Component Model) [50] est un modèle basé sur des composants avec une composition du workflow.

Ce modèle exploite le concept composant-tâche comme unité de base de composition spatio-temporelle. Cette unité est basée sur deux concepts : composant et tâche. Le premier est lié à la composition spatiale et le deuxième représente la notion de composant au niveau de la composition temporelle. STCM est une extension des composants GCM (Grid Component Model) [38] avec des tâches et des ports temporels. Principalement, un composant-tâche encapsule un code et supporte deux types de ports : *ports spatiaux* et *ports temporels*. Comme dans les autres modèles de composants classiques, STCM utilise les ports spatiaux reposant sur le passage de données. De plus, les ports temporels représentent les ports d'entrée et de sortie d'une tâche jouant le rôle d'une fonctionnalité fournie. Ils permettent de définir le type de données acheminées et de déclencher l'exécution d'une tâche avec l'arrivée de données.

Le modèle de composition de STCM est inspiré du langage AGWL [88]. L'idée est de garder la même logique de composition mais introduire le concept de composant-tâche à la place du concept activité. Après la construction d'une application STCM, un composant-tâche a un cycle de vie de son instance contrôlé par ses ports temporels : une instance est créée quand les ports d'entrée sont alimentés par des données et quand ses ports de sortie ne contiennent plus de données cette instance est détruite. Une instance d'un composant-tâche peut avoir plusieurs états comme : *inactif*, *actif*, *s'exécutant* ou *détruit*.

2.6.1.3 FlowVR

Le modèle *FlowVR* [16] est développé par l'équipe MOAIS INRIA Rhône-Alpes à Grenoble et l'équipe PAMDA du LIFO à Orléans. Ce modèle est utilisé et exploité pour produire des applications de simulations parallèles avec un pilotage en temps réel. Il est destiné aux applications distribuées de Réalité Virtuelle sur des clusters ou des environnements de grille.

L'entité de base, appelée un *module* ou un *composant*, est un processus autonome qui peut être parallèle, s'exécutant sur une machine donnée. Il se compose de ports spatiaux permettant de transmettre les données et de ports temporels permettant de transférer les événements sous forme de signaux. Un composant traite les données provenant de ports d'entrée et écrit les données résultats sur les ports de sortie. Un composant n'a pas de vision globale sur la source et la destination des données. Les composants sont itératifs et sont basés sur trois opérations : (1) attendre les données en entrée, (2) récupérer toutes les données en entrée et (3) écrire les données résultats sur les ports de sortie.

L'assemblage des composants se réalise en connectant leurs ports d'entrée avec des ports de sortie en utilisant des connecteurs classés selon trois catégories : connexions directes, connexions anti-saturation et connexions antiblocage [139]. Chaque composant envoie les données résultat dans ces ports de sortie s'il reçoit les données de tous ces ports spatiaux d'entrée avec/sans un signal de déclenchement sur son port temporel d'entrée. Quand il produit des données dans ces ports spatiaux de sortie il envoie un signal sur son port temporel de sortie notifiant la fin d'une itération. La composition forme le graphe de data flow de l'application. Ce graphe se construit grâce à des modules, des liens et des connecteurs élémentaires qui respectent l'ordre d'émission.

Dans FlowVR, la composition se réalise à l'aide des instanciations et la liaison des différents éléments tels que les composants et les connecteurs. Chaque application construite contient deux types de messages. Le premier type représente les données et le deuxième représente les signaux.

Ce modèle peut être utilisé totalement comme (1) un modèle à composition spatiale, (2) un modèle à composition temporelle ou (3) un modèle à composition spatio-temporelle. Dans le premier cas, la structure d'une application se définit en se basant sur des composants indépendamment de tout aspect temporel. Cependant dans le deuxième cas, il est possible de construire des applications avec des liens et des dépendances entre composants. Dans le troisième cas, on combine les deux types de composition en se basant sur les composants (pour la composition spatiale) et des signaux (pour la composition temporelle). Ces signaux permettent d'offrir plusieurs échelles de synchronisation entre composants et la technique *pipelining* qui est utilisée pour optimiser le temps d'exécution d'un processus répétitif.

2.6.2 Analyse & Synthèse

Les systèmes à composition spatio-temporelle permettent d'exploiter les avantages et la complémentarité des deux familles de composition : la composition spatiale et la composition temporelle.

La combinaison de deux compositions spatiale et temporelle diffère d'un modèle à l'autre. ICENI décrit le comportement interne d'un composant avec un formalisme de workflow et il permet de combiner les concepts spatiaux et temporels en exploitant les méta-données. Cela permet d'offrir un plan d'exécution optimal. Cependant, ICENI considère des relations temporelles mais pas d'une façon très explicite et claire entre les composants car ces relations temporelles sont gérées par ICENI mais pas par l'utilisateur.

STCM et FlowVR sont, fondamentalement, des systèmes à base de composants et ils sont basés sur la composition spatio-temporelle. Les composants des deux systèmes contiennent deux types de ports : les ports spatiaux et les ports temporels. Dans STCM et FlowVR, les ports spatiaux permettent de fournir ou utiliser une interface. De plus, les ports temporels permettent d'acheminer le control flow. Dans ces deux systèmes, les paradigmes de la composition sont bien représentés et bien exprimés tels que l'assemblage efficace des différentes parties hétérogènes développés pour produire une application modulaire. Aussi, une réutilisation des composants est bien représentée avec une meilleure lisibilité et une meilleure maintenance.

2.7 LA RECONFIGURATION DES ARCHITECTURES PAR COMPOSANTS

Cette section présente la reconfiguration dynamique des systèmes basés sur des composants. La reconfiguration dynamique est une exigence importante pour les systèmes modernes et permet de modifier l'architecture des systèmes durant leur exécution afin d'appliquer, par exemple, des corrections et des mises à jour où des composants et des connecteurs d'un système peuvent être insérés, supprimés ou remplacés. La reconfiguration dynamique présentée est classée selon deux catégories qui vont être décrites par la suite : la *reconfiguration centralisée* et la *reconfiguration auto-adaptative*.

2.7.1 La reconfiguration centralisée

La reconfiguration centralisée des approches par composants se base sur une entité qui se charge de réaliser une reconfiguration donnée. Cette entité peut être un composant spécial ou un agent permettant de gérer la reconfiguration d'une façon centrale. Ce type de reconfiguration a été largement étudiée et il représente un élément important dans de nombreux contextes où l'application ne devrait pas s'arrêter. Dans le contexte des services Web, il existe plusieurs approches pour effectuer une reconfiguration dynamique centralisée. Par exemple dans [96], un framework est présenté qui permet la conception de processus WS-BPEL de façon modulaire basée sur des modèles réutilisables. Ce framework utilise un contrôleur pour sélectionner les définitions

des services Web à partir une bibliothèque et les intégrer dans un processus maître. Ce dernier contient tous les détails spécifiques de tous les scénarios possibles des services Web utilisés, il est destiné aux concepteurs pour pouvoir modéliser les fonctionnalités dynamiquement et indépendamment des implémentations concrètes. Dans un autre travail [196], la reconfiguration a été traitée en utilisant plusieurs agents répartis pour reconfigurer les parties participantes pour fournir une application fiable, sécurisée et interopérable. Ces agents distribués effectuent les diverses vérifications d'exécution, d'audit et de reconfiguration. Une des propriétés intéressantes de l'approche proposée est qu'elle est conçue pour réaliser les modifications facilement et ces changements sont vérifiés et appliqués à l'exécution.

Dans un autre contexte, concernant les applications industrielles basées sur le modèle FRACTAL, une approche a été proposée dans [95] permettant de gérer la reconfiguration de ces applications en utilisant la solution *HyperManager*. Cette solution n'est qu'une autre application qui est capable de surveiller une application distribuée et d'effectuer une telle reconfiguration qui est basée sur les deux événements suivants : *pull* et *push*. Une autre approche basée sur le travail présenté dans [126] intègre un gestionnaire de reconfiguration qui est le responsable de l'exécution des reconfigurations. Ce gestionnaire a la connaissance de différents changements à venir et sa tâche principale est de les exécuter. Il contrôle aussi l'état du composant afin d'assurer une reconfiguration sûre. Cette approche effectue une reconfiguration structurelle pour modifier la configuration des applications distribuées en cours d'exécution. Une configuration se compose de nœuds de traitement interconnectés en utilisant des liens de communication bidirectionnels. Lors d'une modification, le gestionnaire de reconfiguration traite les nœuds de traitement directement impactés par le changement et les nœuds directement adjacents à eux pour les mettre dans un état de repos (quiescent state). Dans cet état, il est prévu qu'un nœud ne va jamais recevoir des données au cours de la reconfiguration. En outre, pour les applications basées sur la norme IEC 61499 présentée dans la section 2.4.2, le travail de [125] met l'accent sur le problème de la reconfiguration des systèmes de contrôle multi-agents embarqués distribués. Au niveau de l'approche proposée, une architecture de multi-agents est proposée dans laquelle un agent de reconfiguration est affecté à chaque dispositif de l'environnement d'exécution pour appliquer la reconfiguration locale, et un agent de coordination pour coordonner entre les dispositifs afin de garantir une reconfiguration distribuée cohérente. Dans le même contexte, une reconfiguration en temps réel des composants est présentée dans [114]. Elle est gérée centralement à l'aide d'un (1) composant gestionnaire qui est chargé de contrôler le processus de reconfiguration. Il choisit quelle reconfiguration doit être exécutée selon la situation. (2) Un composant exécuteur qui encapsule les règles de reconfiguration et est responsable de leur exécution. Dans [83], une logique de schéma temporel linéaire a été définie pour exprimer des propriétés de reconfiguration dynamique dans un modèle à base de composants. Puis, un moniteur repose sur cette formalisation pour procéder et pour contrôler la reconfiguration et si nécessaire pour revenir à un état de sécurité.

2.7.2 La reconfiguration auto-adaptative

Les applications basées sur la reconfiguration auto-adaptative sont capables de gérer leur changement dynamiquement et d'une façon autonome. Ce type de reconfiguration est utilisée au niveau du modèle Fractal [73], présenté dans la section 2.4.1, en proposant son extension nommée SAFRAN pour le développement de l'aspect d'adaptation comme politiques d'adaptation réactives. Ces politiques détectent les évolutions du contexte d'exécution et adaptent l'application en la reconfigurant. De cette façon, SAFRAN permet le développement modulaire de politiques d'adaptation et leur tissage dynamique dans les applications en cours d'exécution. En effet, SAFRAN permet de développer des applications auto-adaptatives basées sur le modèle de composants Fractal et est conçu autour de trois grands principes : (1) l'utilisation d'un modèle de composant dynamique, c-à-d Fractal, pour construire des applications qui peuvent être adaptées à l'exécution. (2) L'utilisation de concepts et de techniques AOP (Aspect-Oriented Programming) pour développer la logique de l'adaptation séparément à partir du code métier et de les tisser dynamiquement pour produire des applications auto-adaptatives. (3) Enfin, l'utilisation d'un langage dédié (Domain Specific Language [199]) pour exprimer cette logique d'adaptation.

Dans une architecture orientée services (service oriented architecture, SOA), une approche nommée MoDAR (Model-Driven Development of DASS with Aspects and Rules) est proposée [217] pour le développement des systèmes basés sur les services adaptatifs dynamiquement (dynamically adaptive service-based systems, DASS) en utilisant des règles et des aspects. De plus, cette approche est adoptée pour faciliter leur développement dans la mesure d'adapter dynamiquement leurs comportements en fonction des changements d'exigences fonctionnelles. Dans cette approche, une fonctionnalité d'un système se modélise en deux parties : (1) une partie stable appelée le modèle de base décrite en utilisant des processus métiers. (2) Une partie volatile appelée le modèle à variable décrite à l'aide de règles métier. La méthodologie de cette approche se base sur une séparation de la partie variable et de la partie processus stable et sur une modélisation de la partie variable comme des règles métiers avec un tissage de ces règles dans un processus de base. En effet, le système cible est adaptatif dynamiquement dans le sens que les règles de la partie variable d'un système peuvent changer à l'exécution sans affecter le processus de base.

Le modèle CompAA (Auto-Adaptable Components) permet de faire face à l'adaptation dynamique qui traite l'évolution constante des systèmes c-à-d l'évolution en termes de disponibilité des services, des mises à jour, les erreurs ou l'évolution des besoins à l'exécution [131]. Ce modèle produit des systèmes avec des propriétés les plus dynamiques et les plus autonomes par rapport la découverte des services disponibles et leur composition. L'idée est de combiner des composants et des agents dans une approche mixte. L'utilisation des composants est motivée par les résultats obtenus, d'une part dans le domaine de la réutilisation, et d'autre part dans le domaine de la modélisation et du déploiement d'applications distribuées. Puisque les composants sont limités en termes de flexibilité et d'autonomie, les agents sont utilisés pour automatiser les applications.

2.7.3 Analyse & Synthèse

La différence entre une reconfiguration centralisée et une reconfiguration auto-adaptative c'est que la première s'effectue en se basant sur une entité centrale pas forcément automatique pour réaliser une modification dans la structure d'une application. Mais, la deuxième s'effectue d'une façon autonome pour la gestion des applications étudiées. Ce type de reconfiguration permet aux applications de s'organiser, pendant son exécution, sans aucune intervention externe et de gérer leur configuration c-à-d être conformes à ses besoins, être sûrs d'atteindre un état stable, la détection des erreurs avec leur réparation et l'adaptation autonome des paramètres. Les deux types de reconfiguration présentées se différencient par rapport la complexité de changement de la structure d'applications, les types des communications utilisées et la façon de gérer les informations.

2.8 CONCLUSION

Nous avons étudié des modèles, parmi les plus connus, à base de composants logiciels et leurs différents éléments architecturaux pour construire des applications basées sur la composition spatiale. À la fin, nous avons réalisé une comparaison entre les modèles présentés. Cette comparaison a montré que la composition spatiale se base sur le paradigme client-serveur entre les composants [51]. Nous parlons dans ce cas, de l'invocation des opérations d'un composant serveur à partir d'un composant client. Puisque la composition spatiale est seulement une relation entre composants au niveau de leurs ports, qui sont compatibles, elle n'apporte pas d'informations concernant l'ordre d'exécution des composants. En effet, l'absence de la temporalité explicite rend la composition spatiale limitée. Cela implique une difficulté pour maîtriser le comportement d'une application. Cette composition ne permet pas d'optimiser le déploiement d'une application, par exemple, dans le sens de l'ordre d'exécution de ses composants contrairement à la composition temporelle. Cette dernière vise à définir une relation explicite de dépendances entre les composants. La composition temporelle possède trois formalismes pour décrire une telle relation *workflow*, *data flow* et *control flow*. Le premier décrit les différentes formes des flux, le deuxième présente une dépendance par rapport à la disponibilité des données. Par contre, le troisième formalisme définit l'ordre d'exécution par des structures de contrôle telles que les séquences, les boucles, les sauts inconditionnels, les sauts conditionnels, etc. D'autre part, la composition spatio-temporelle repose sur les deux points forts suivants : (1) le couplage fort de la composition spatiale et (2) l'optimisation d'utilisation des ressources et le déploiement de la composition temporelle.

À cause de la nature hétérogène des applications de la visualisation scientifique interactive présentées dans la section 1.3.2, la conception de ces applications a besoin d'une séparation des préoccupations pour les spécialistes et les non spécialistes. De plus, la performance est un critère majeur pour les scientifiques lors de la production

de leurs applications. Ces applications alors peuvent contenir des éléments fonctionnant à des fréquences très différentes, par exemple, un composant de simulation émet des données à une fréquence vraiment très élevée par rapport à un composant de visualisation. En outre, la synchronisation entre composants des applications de la visualisation scientifique interactive joue un rôle important pour avoir une meilleure performance.

Afin de garantir ces besoins, la composition spatio-temporelle est capable de fournir une séparation entre les différentes parties des applications visées comme l'interaction, la simulation et la visualisation. En plus, ce type de composition exploite bien les avantages de la programmation par composants tels que le control flow, le data flow, l'acheminement de données, le passage des événements, la gestion déterministe du cycle de vie des composants, l'abstraction, la flexibilité, la réutilisation, etc. En outre, il est habilité à externaliser et représenter explicitement le control flow par rapport au data flow :

- pour bien représenter les schémas de coordination et la séparation entre les codes fonctionnels et non fonctionnels ;
- pour gérer la communication et la redistribution de données entre les composants.

Une fois l'application de la visualisation scientifique interactive est construite, il est intéressant de la reconfigurer dynamiquement durant son exécution pour faire face à des évolutions, par exemple, de leur contexte de fonctionnement ou des besoins des utilisateurs comme l'ajout des étapes d'analyse visuelle. La gestion de la reconfiguration dynamique doit tenir compte de toutes les spécificités d'une application interactive haute performance composée de codes très hétérogènes.

Un tel modèle proposé basé sur la composition spatio-temporelle et appliqué à la visualisation scientifique interactive doit être bien formé et spécifié pour construire et reconfigurer dynamiquement les applications de simulation scientifique et d'analyse visuelle. Le chapitre suivant présente une étude de différentes méthodes formelles qui ont comme but l'utilisation d'un raisonnement mathématique pour concevoir et réaliser des applications valides.

LA FORMALISATION DES MODÈLES

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the universe trying to build bigger and better idiots. So far, the universe is winning.

Rick COOK

SOMMAIRE

3.1	INTRODUCTION	53
3.2	L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES	54
3.2.1	UML	55
3.2.2	Méthode B	56
3.3	LA SIMULATION ET LE TEST POUR LA VÉRIFICATION	57
3.3.1	Vérification par la simulation	57
3.3.2	Vérification par le test	58
3.4	LES MÉTHODES FORMELLES	58
3.4.1	Définitions	58
3.4.2	Model Checking	59
3.4.3	SAT	61
3.4.4	Les algèbres de processus	61
3.4.5	Les réseaux de Petri	63
3.5	CONCLUSION	64

3.1 INTRODUCTION

L'évolution exponentielle des nouvelles technologies matérielles ou logicielles, joue un rôle moteur primordial dans la communication, l'accès aux sources d'information, le stockage, la manipulation, la production et la transmission d'informations. En effet, la demande de logiciels perfectionnés a augmenté et est devenue importante en imposant des contraintes comme l'alourdissement du développement. La productivité des informaticiens et le temps de développement deviennent critiques. Ces derniers rencontrent plusieurs problèmes face à la nature des applications, à leur taille ou à leurs environnements distribués/hétérogènes.

Pour surmonter ces difficultés, les informaticiens ont besoin d'utiliser des méthodes formelles d'analyse et de conception. Ces méthodes permettent de décrire les propriétés d'une application en fournissant des représentations :

- apportant des sémantiques claires et non ambiguës basées sur des principes mathématiques ;
- proposant une abstraction de haut niveau ;
- conduisant à des descriptions précises ;
- permettant de démontrer des propriétés.

Pour mieux concevoir un système et assurer sa fiabilité, il est nécessaire d'utiliser des techniques de vérification. Ces techniques sont donc devenues incontournables dans le développement pour repérer des erreurs sans accorder plus de temps à la vérification qu'à la conception des systèmes. Une telle vérification garantit le bon fonctionnement et les mesures de performances en apportant des preuves sur l'absence ou la présence de comportements spécifiques pour valider ou refuser un modèle.

Des travaux de recherches ont été menés afin de construire correctement ces systèmes par des méthodes telles que les méthodes de *l'ingénierie dirigée par les modèles*. Ces méthodes contribuent à l'amélioration de modèles, concepts, langages et à la fois le problème posé c-à-d le besoin et sa solution. Dans d'autres travaux, la *vérification* est utilisée comme méthode pour étudier et vérifier les systèmes en exploitant des techniques telles que la *simulation* ou le *test*. De plus, autres travaux de recherches présentent et étudient des méthodes formelles telles que le *Model Checking*, *SAT*, les *algèbres de processus* et les *réseaux de Petri*.

Dans ce chapitre, nous présentons l'ingénierie dirigée par les modèles qui traite la conception des systèmes correctes par construction avec une vérification pour éviter la présence d'erreurs logiques. À cet égard, la vérification de modèle contribue à améliorer la fiabilité et la robustesse des systèmes. Le but est d'étudier l'absence de comportements indésirables avec une modélisation des comportements attendus des systèmes et particulièrement des applications distribuées. En outre, l'intégration aux systèmes des méthodes formelles facilite la preuve des propriétés, et par conséquent la validation des applications produites.

Nous commençons par présenter la définition du domaine de l'ingénierie dirigée par les modèles avec une présentation de quelques méthodes de modélisation utilisées. Ensuite, la vérification par la simulation et la vérification par le test seront présentées. Enfin, des méthodes formelles seront introduites et nous allons présenter intuitivement dans la conclusion le modèle formel que nous allons utiliser pour modéliser et formaliser la sémantique de notre approche par composants.

3.2 L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES

L'Ingénierie Dirigée par les Modèles (IDM) est un domaine basé sur des approches formelles de la vérification et de la validation [183]. Dans ces approches la vérification concerne les phases de création et la validation concerne les phases de recette ou de test d'acceptation. L'IDM consiste à manipuler différents modèles de l'application à produire, depuis une description très abstraite jusqu'à une représentation qui correspond à l'implantation effective du système. En effet, l'IDM conçoit l'intégralité du

cycle de l'application et permet de la produire correctement par construction en se basant sur un raffinement itératif d'une spécification abstraite. L'IDM consiste à décrire séparément les modèles au niveau des différentes phases du cycle de développement d'une application (figure 3.1). Précisément IDM préconise la constitution des modèles :

- de besoins fonctionnels et techniques ;
- d'analyse et de conception ;
- de déploiement et de code.

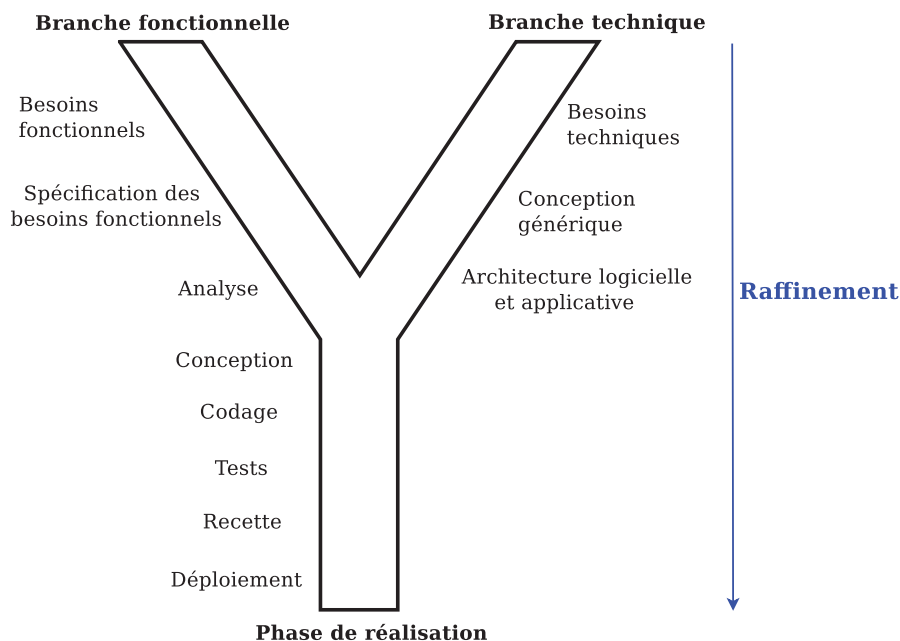


FIGURE 3.1 – Le processus Y dirigé par les modèles.

La modélisation dans l'IDM se base souvent sur les standards de l'OMG (Object Management Group) et en particulier le langage de modélisation UML (Unified Modeling Language).

3.2.1 UML

UML est une notation graphique conçue pour représenter, spécifier, construire et documenter les systèmes logiciels. En 1997, UML est devenu une norme OMG. Ses deux principaux objectifs sont la modélisation des systèmes utilisant les techniques orientées objets, depuis la conception jusqu'à la maintenance, et la création d'un langage abstrait compréhensible par l'homme et interprétable par les machines [63]. Il est basé sur un langage expressif en utilisant plusieurs notations permettant de représenter le même système selon différentes vues.

UML représente un vrai support de communication entre les différents intervenants d'un système facilitant la représentation et la compréhension de plusieurs éléments à savoir les activités des acteurs, des processus, des schémas de base de données

et des composants logiciels. Le langage UML est simple à comprendre et fournit un bon moyen de communication entre différentes entités au sein d'un projet. En effet, il fixe des règles de mise en œuvre décrivant la jonction entre les différents points de vue statique, dynamique et fonctionnel, l'enchaînement des actions, l'ordonnancement des tâches et la répartition des responsabilités. UML apporte une modélisation graphique à base de diagrammes, classés selon trois catégories : structurel, comportemental et interaction. UML a été largement mis en pratique, dans plusieurs domaines, pour modéliser les services Web et leur composition [106, 182]. De plus, il est utilisé pour la modélisation des systèmes embarqués [64] et pour la vérification de ces derniers en temps réel [119].

Actuellement un grand intérêt est porté à l'enrichissement de la modélisation à base d'UML. Cet intérêt est motivé principalement par l'apport d'analyse formelle aux systèmes modélisés par UML. Il est à noter qu'il existe des efforts de recherche focalisant sur l'utilisation des méthodes formelles avec UML comme la méthode B. Cette méthode couvre toutes les phases d'un cycle de développement formel, partant d'une spécification abstraite de haut niveau et qui par raffinement aboutit à un code exécutable. Cette approche est basée sur la preuve de théorèmes et propose des méthodes de vérification de propriétés.

3.2.2 Méthode B

La méthode B¹ est une méthode de spécification formelle et une approche par modèle abstrait définissant un modèle mathématique du comportement d'un système. Cette approche est constructive permettant de définir une structure de données et un ensemble d'opérations. Elle est basée sur la logique du premier ordre et la théorie des ensembles. Cette méthode effectue une analyse rigoureuse et prouve automatiquement si des propriétés sont cohérentes et non contradictoires à l'aide des preuves mathématiques.

L'objectif de la méthode B est de fournir une technique et des outils capables d'aider la construction de logiciels d'une manière plus sûre. C'est une méthode transformationnelle c-à-d le modèle abstrait (une machine abstraite) se transforme étape par étape (par des raffinements), jusqu'à l'obtention d'un modèle concret, proche du code source (l'implémentation).

La méthode B a été conçue pour modéliser les systèmes distribués et plus généralement des systèmes à événements discrets en utilisant son extension qui est la méthode Event-B². Cette dernière fournit aussi une description de systèmes à base d'événements et de transitions avec une preuve de propriétés temporelles. Event-B simplifie la notation de la méthode B, ce qui la rend plus facile à apprendre et à utiliser. En effet, elle est constituée d'une description d'états avec des constantes, des variables, des invariants et des événements remplaçant les opérations de la méthode B. Cette

1. <http://www.methode-b.com/>

2. <http://www.event-b.org/>

méthode permet de décrire, par exemple, une évolution indéterministe d'un système pour déterminer plusieurs propriétés comme la détection de blocages.

La méthode B est utilisée dans des projets industriels. Par exemple, en France, les exigences fonctionnelles du système SACEM présent dans ligne A du RER à Paris ont été formellement construites avec la méthode B [107]. De plus, elle est adoptée dans le système de métro automatique de la ligne 14, qui représente la première ligne de métro automatique à Paris [39].

3.3 LA SIMULATION ET LE TEST POUR LA VÉRIFICATION

La vérification est le processus consistant à vérifier si une conception satisfait certaines spécifications (propriétés). La vérification peut traiter :

1. la vérification de la correction : aucune erreur de conception ;
2. la fiabilité : le système fonctionne tout le temps ;
3. la sécurité : pas d'utilisation non-autorisée.

La vérification est utilisée pour éviter la présence d'erreurs logiques. À cet égard, la vérification de modèle contribue à améliorer la performance des systèmes distribués et aussi à améliorer le niveau de confiance que nous pouvons mettre en eux. Les méthodes de vérification peuvent se classer en *vérification par la simulation* et *vérification par le test*.

3.3.1 Vérification par la simulation

Les modèles de simulation sont de plus en plus utilisés pour résoudre des problèmes, pour aider à la prise de décision et pour savoir si un modèle et ses résultats sont corrects. Dans le cadre de la vérification, la simulation d'un modèle est le processus de confirmation qu'il est correctement mis en œuvre par rapport au modèle conceptuel [32].

Lors de la vérification par la simulation, le système est testé pour détecter et corriger les erreurs dans la mise en œuvre du système. Divers procédés et techniques sont utilisés pour assurer que le système correspond à des spécifications et à des hypothèses à l'égard du modèle. L'objectif de la vérification par la simulation du modèle est d'assurer que sa mise en œuvre est correcte.

Pratiquement, étant donné un modèle de système et une spécification désirée, les concepteurs de systèmes reposent sur l'analyse et la simulation pour étudier le comportement de ce système. Les approches fondées sur la simulation assurent qu'un nombre fini de traces d'exécution ou de scénarios du système établi par l'utilisateur répond à la spécification désirée. La vérification en utilisant la simulation ne se base pas sur un seul cas d'exécution mais sur tous les cas d'exécution possibles pour montrer le bon fonctionnement du système. Cependant, il est difficile de générer tous les cas d'exécution possibles ce qui peut empêcher de détecter les erreurs cachées au niveau des scénarios non déroulés [100]. Aussi, la vérification par la simulation implique une lenteur du processus de vérification.

3.3.2 Vérification par le test

La vérification par le test est une technique qui nécessite une préparation de tests à l'avance avec une description de ses principales phases des spécifications précises du système. Cette vérification est effectuée dans le cas où c'est difficile d'établir un modèle du système [155]. Elle se base sur une construction des scénarios puis leur exécution afin de valider ou vérifier le bon ou mauvais fonctionnement du système.

Deux types de tests existent, tests de défauts et tests de validation. Le premier peut être effectué pour détecter les défauts et les erreurs dans un système. Un test de défauts valide implique la découverte d'un défaut dans le système. Tandis que dans le test de validation, la détection des défauts s'effectue pour montrer que le système construit n'est pas correct et n'est pas utilisable en pratique. De plus, un test valide montre que le scénario exécuté est bien implémenté. Avec le test, la vérification pour découvrir des erreurs ou/et prouver sa correction peut s'exécuter manuellement ou automatiquement.

Les tests peuvent être effectués sur le système en tant que test à boîte noire pour tester une application en vérifiant que les sorties obtenues sont bien celles prévues pour les données d'entrée [1]. Dans ce cas le système testé n'est pas visible. Par contre dans le test à boîte blanche, l'accès total à la structure interne de l'implémentation du système est autorisé et le test est effectué avec des techniques exhaustives [101]. La vérification basée sur le test ne fournit pas une vraie garantie pour vérifier les systèmes par rapport à une certaine spécification ou une propriété formelle, en utilisant par exemple des méthodes formelles qui seront présentées dans la section suivante.

3.4 LES MÉTHODES FORMELLES

3.4.1 Définitions

Une méthode formelle permet de spécifier et modéliser les comportements désirés et les propriétés structurelles d'un système afin de répondre à des objectifs définis, comme décrit par Hoare dans [116] :

« A scientific theory is formalised as a mathematical model of reality, from which can be deduced or calculated the observable properties and behaviour of a well-defined class of processes in the physical world. »

De plus, une méthode formelle est une technique permettant de raisonner rigoureusement sur des logiciels informatiques pour démontrer leur validité par rapport à des spécifications. La définition proposée par Wing dans [212] décrit plus clairement l'utilité d'une méthode formelle :

« Applied to computer systems development, formal methods provide mathematically based techniques that describe system properties. As such, they present a framework for systematically specifying, developing, and verifying systems. »

L'analyse formelle, donc, est une méthode pour prouver la logique d'un système complexe, pour l'analyser et pour le vérifier.

Son utilisation a de nombreux avantages [110]. En voici un bref aperçu :

- les méthodes formelles sont puissantes pour détecter les erreurs et éliminer certaines classes d'erreurs ;
- elles fonctionnent, en grande partie, en forçant les développeurs à réfléchir profondément aux systèmes à concevoir ;
- elles sont basées sur des notations mathématiques qui sont faciles à comprendre par rapport aux programmes ;
- elles peuvent diminuer le coût de développement.

Au sens large, les méthodes formelles sont des outils puissants qui ne diffèrent pas uniquement au niveau des notations mais aussi au niveau du choix du domaine de la sémantique et le niveau d'analyse des propriétés des systèmes. L'intégration de ces méthodes dans le processus de développement des applications complexes s'effectue notamment dans le Model Checking, SAT, les algèbres de processus et les réseaux de Petri.

3.4.2 Model Checking

Le Model Checking a émergé comme une approche prometteuse pour automatiser la vérification si un modèle de système répond à ses spécifications [65]. Ce modèle a le même impact que l'exécution d'un test exhaustif, en utilisant des évaluations symboliques où chaque scénario possible est vérifié pour la correction. Le Model Checking utilise plusieurs techniques de vérification automatiques pour des systèmes à états finis et des systèmes complexes tels que les applications de conceptions de circuits séquentiels et les applications des protocoles de communication. Ce type de vérification est utilisé dans plusieurs travaux et particulièrement au niveau des modèles à base de composants [120, 163, 201]. Par exemple dans [28], le Model Checking est introduit dans la plateforme *Vercors* pour vérifier l'exactitude du modèle par rapport à un ensemble de propriétés qui représentent des exigences de l'utilisateur.

Le Model Checking peut être caractérisé comme une procédure d'inférence qui décide si une structure M satisfait une spécification donnée ϕ , formulée comme suit : $M \models \phi$. Pratiquement, un algorithme de Model Checking prend en entrée une abstraction du comportement d'un système de transitions et une formule d'une certaine logique, et retourne si l'abstraction satisfait ou non la formule.

Le Model Checking est une méthode de vérification efficace et complètement automatique par rapport aux méthodes basées sur la simulation et le test. Il est basé sur des techniques d'abstractions permettant de remédier aux problèmes de temps et de disponibilité des ressources tout en fournissant une vérification complète et efficace. Ce modèle peut être basé sur la description à base de structure de *kripke* [56] de tous les états possibles du système, et les transitions entre les états et les propriétés vérifiant chaque état. La figure 3.2 montre le principe du Model Checking. D'abord, le modèle

est décrit et ses états sont représentés, par exemple, par une structure de graphe. Ensuite, la propriété désirée est vérifiée pour la correction dans la structure de graphe. Cette propriété désirée attendue peut être exprimée dans le temps à l'aide de la *logique temporelle linéaire* (Linear Temporal Logic, LTL). Cette dernière est une logique propositionnelle avec des opérateurs supplémentaires pour le temps. En effet, elle est une logique temporelle modale avec des modalités qui se réfèrent au temps [171].

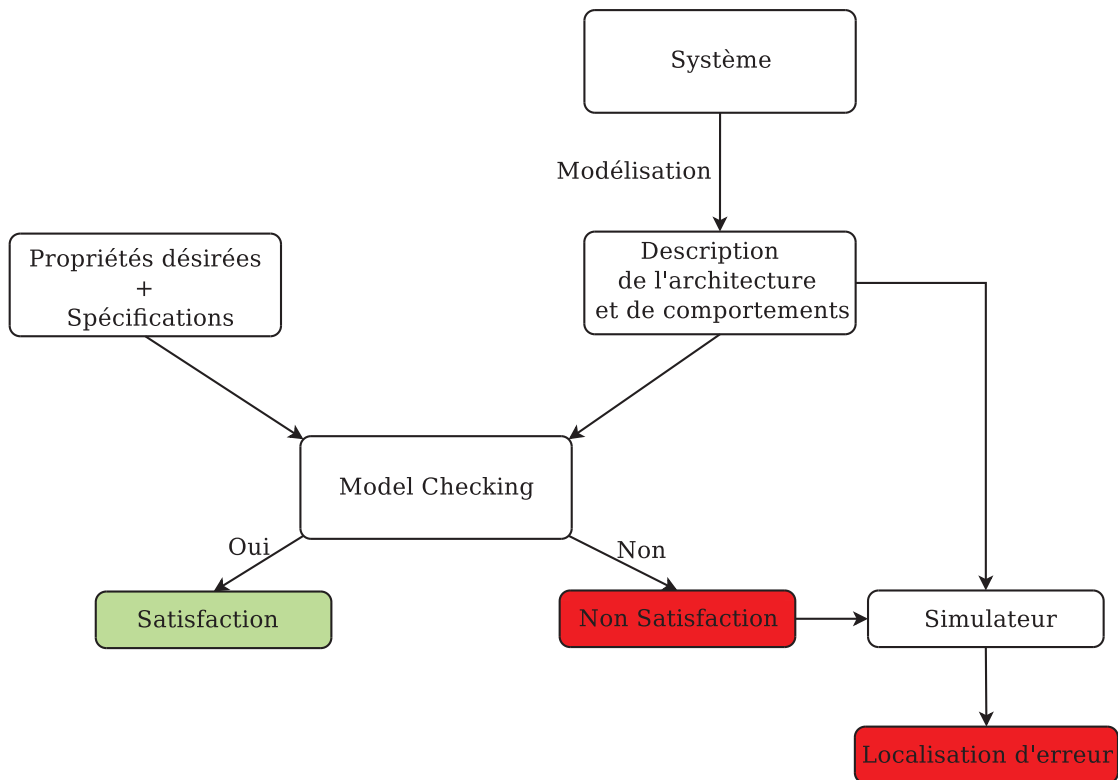


FIGURE 3.2 – Le Principe du Model Checking.

Finalement, si la propriété, par exemple exprimée par LTL, n'est pas vérifiée, un contre-exemple peut être fourni pour identifier un cas d'exécution non conforme et ainsi pour reproduire un chemin menant à l'erreur donnée. Ce principe se répète jusqu'à ce que toutes les propriétés soient vérifiées par le modèle.

Cependant, l'énumération de tous les états conduit à une explosion combinatoire pouvant compliquer la vérification en utilisant des méthodes formelles classiques. Il s'agit donc de mettre en place des techniques spécifiques de réduction de l'espace des états de ce problème pour fournir une vérification garantie et efficace. Une telle réduction est proposée en utilisant par exemple des approches formelles spécifiques comme SAT, les algèbres de processus et les réseaux de Petri.

3.4.3 SAT

Le problème SAT est un problème se basant sur la logique propositionnelle. Il consiste à décider si une formule mise sous forme normal conjonctive (Conjunctive Normal Form, CNF) admet un modèle ou non. Cette forme de représentation considère la donnée d'une conjonction de clauses où une clause est une disjonction de littéraux. Toutefois il existe des transformations permettant de mettre une formule propositionnelle quelconque en CNF. Le problème SAT occupe une place particulière en théorie de la complexité puisqu'il est le premier à être montré NP-Complet par Cook [67].

Depuis les années soixante, de nombreux algorithmes de résolution de SAT ont vu le jour [75, 76]. La recherche dans ce domaine a permis l'avènement de solveurs appelés solveurs SAT modernes ou CDCL [154, 222] capables de résoudre des problèmes contenant des millions de variables et de clauses. Cette efficacité de résolution a permis à cette technologie d'être exportée vers d'autres domaines d'applications. En effet, plusieurs problèmes issus de la planification classique, la cryptographie, la configuration de produits, etc. ont été encodés vers SAT.

Le champ d'application de SAT a été également élargi à d'autres domaines de recherche comme le problème des formules booléennes quantifiées (Quantified Boolean Formula, QBF), la satisfiabilité maximale (MaxSAT) ou encore dans le comptage et l'énumération de tous les modèles d'une formule CNF.

3.4.4 Les algèbres de processus

Les algèbres de processus sont une famille de langages formels permettant de modéliser les systèmes concurrents ou distribués. Ces algèbres fournissent un formalisme théorique sur lequel s'appuient divers outils de spécification et de vérification [23, 46, 49]. Ils traitent des expressions formées de constantes, de variables et d'opérateurs qui représentent des processus. Dans les algèbres de processus les systèmes sont décrits par les équations qui expriment leur comportement et/ou leurs états. En plus, les systèmes de processus communicants sont représentés par des expressions algébriques, nommées des expressions de comportement. Plusieurs algèbres ont été étudiées dans la littérature comme :

- CSP (Communicating Sequential Processes) [117] : est un langage formel pour décrire les modes d'interaction dans les systèmes concurrents. Il permet la description des systèmes en termes de processus de composants qui fonctionnent et évoluent indépendamment du reste de l'univers et interagissent avec d'autres processus en se basant sur le passage de messages via des canaux. Ce langage est appliqué au niveau des systèmes critiques, par exemple, les systèmes de commandes des avions et au niveau des protocoles de réseau local [74];
- CCS (Calculus of Communicating Systems) [152] : fournit un outil pour une description de haut niveau des interactions, communications et synchronisations entre un ensemble d'agents ou des processus indépendants. Ce langage modélise les communications indivisibles entre deux participants en exploitant

des primitives pour décrire la composition parallèle et le choix entre les actions. Dans ce langage, les expressions sont interprétées comme un système de transition étiqueté. Il est utile pour évaluer la vérification de la correction qualitative des propriétés d'un système tel que les blocages ;

- LOTOS (Language Of Temporal Ordering Specification) [121, 197] : est un standard FDT (Formal Description Technique) qui a pour objectif la spécification des communications des systèmes distribués. Il se base sur un langage d'abstraction de haut niveau et une base mathématique, adaptés à la spécification et l'analyse des systèmes complexes. LOTOS se compose de deux sous-langages intégrés pour spécifier les types de données ADT (Abstract Data Types) et les comportements (Process Algebra). Il est utilisé pour analyser et spécifier une variété de systèmes et il est supporté par des outils pour la spécification, la simulation, la compilation, la génération de test et la vérification formelle. LOTOS est utilisé pour la spécification de protocole dans les normes ISO OSI [47] ;
- π -calcul [153, 177] : est un langage destiné à modéliser les échanges de messages entre des programmes parallèles et raisonner sur des systèmes distribués communicants. Ce langage se focalise sur la migration de processus entre pairs, l'interaction des processus via des canaux dynamiques et la communication des canaux privés. De plus, il décrit des systèmes de calculs concurrents dont la configuration réseau peut varier au cours du temps. Il apporte la mobilité qui est un concept important qui n'est pas modélisé par le langage CCS. Cette mobilité signifie que les processus peuvent se déplacer dans l'espace physique des sites de calcul, ils peuvent se déplacer dans l'espace virtuel des processus liés et les liens peuvent se déplacer dans l'espace virtuel des processus liés. Le π -calcul est utilisé, par exemple, pour la description et l'analyse des protocoles de sécurité [7] et dans la biologie moléculaire pour modéliser, analyser et simuler les systèmes biologiques [168]. En outre, ce langage est appliqué pour modéliser et vérifier la composition des services Web [8, 215].

Classiquement, les algèbres de processus fournissent une analyse par bisimulation (resp. simulation), à savoir, nous pouvons déterminer si deux processus ont des comportements équivalents. Deux processus sont considérés comme équivalents s'ils présentent les mêmes traces d'exécution. Un processus est contenu dans un autre si l'ensemble de ses traces d'exécution sont inclus dans l'ensemble des traces d'exécution de l'autre. De plus, deux processus sont équivalents s'ils ont des arbres d'exécution équivalents c-à-d ils se simulent. Un processus est simulé par un autre si tous ses comportements sont contenus dans les comportements de l'autre.

Par exemple, ce type d'analyse est utilisé dans [33] pour vérifier une application à base de composants FRACTAL en utilisant la plateforme *VerCors*. Dans ce cas, cet analyse est utilisé pour vérifier la spécification du modèle par rapport à un ensemble de propriétés de logique temporelle et de fournir des diagnostics pour le concepteur. La détection de blocages est une propriété parmi les propriétés vérifiées.

L'analyse par bisimulation est aussi utile, par exemple, pour montrer si un processus peut substituer un autre processus d'un système. D'autres exemples d'utilisation

de la bisimulation sont la vérification de la redondance de processus dans un système et l'étude des propriétés de sûreté.

3.4.5 Les réseaux de Petri

Un réseau de Petri est un modèle mathématique servant à représenter divers systèmes et applications informatiques et a été conçu en 1962 par *Carl Adam Petri* dans sa thèse intitulée « Communication avec Automates » [167]. Un réseau de Petri constitue un langage de modélisation orienté états pour simuler et vérifier certains ensembles de propriétés comportementales des systèmes. Un réseau de Petri permet de représenter différents états possibles au sein d'un système et de représenter les événements qui déclenchent le changement d'un état [123]. Un réseau de Petri est un graphe bipartite et s'appuie sur des *places*, des *transitions* et des *arcs*. Un arc relie alternativement une place à une transition et une transition à une place. Les places jouent le rôle de variables d'états. Elles prennent des valeurs représentées par des marques appelées *jetons* et qui permettent de représenter les ressources disponibles. Les transitions représentent les actions pouvant exécuter et effectuer des échanges de jetons entre les places. En effet, ces transitions représentent des événements dont l'occurrence provoque des changements d'états. Les arcs représentent l'orientation des échanges de jetons.

Les réseaux de Petri sont souvent utilisés pour modéliser formellement et graphiquement les systèmes dynamiques échangeant les ressources. Ils ont fait l'objet de nombreux travaux, par exemple, dans les systèmes de santé pour modéliser, à l'aide d'une nouvelle classe, les protocoles médicaux et les ressources dont ces systèmes ont besoin pour progresser [143]. De plus, dans les systèmes de fabrication flexibles pour étudier la prévention de blocages en utilisant une modélisation de ces systèmes concurrents à l'aide des réseaux de Petri [195].

Particulièrement, les réseaux de Petri sont aussi utilisés pour modéliser et étudier des systèmes à base des architectures par composants [35, 71, 141, 218]. Par exemple, dans [112] ils sont utilisés pour spécifier la contrôlabilité et la substituabilité des protocoles de services qui interagissent de manière asynchrone et également dans [20, 61, 79, 190, 216], ils sont utilisés pour modéliser la composition des services Web. Il existe de nombreuses extensions de réseaux de Petri représentant des modèles comme le montre la figure 3.3.

La modélisation des systèmes complexes a prouvé que tous les types de réseaux de Petri ne possèdent pas la même importance concernant :

- la définition et la modélisation des systèmes parallèles, distribués et synchrones ;
- la compréhension et l'étude des concepts qualitatifs et quantitatives, pour effectuer, par exemple, une telle simulation ;
- le processus d'analyse des comportements dans le but de réaliser, par exemple, le Model Checking.

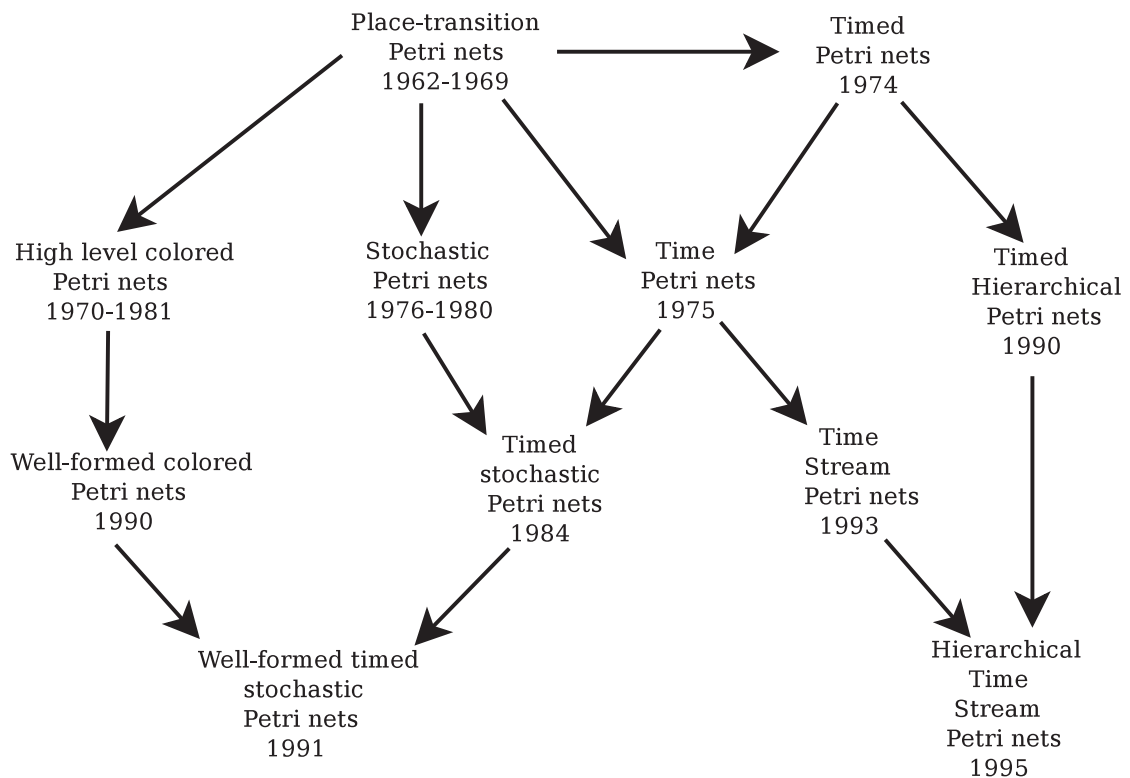


FIGURE 3.3 – Les modèles principaux de réseaux de Petri [81].

3.5 CONCLUSION

Nous avons souligné dans ce chapitre la définition fondamentale d'une méthode formelle et sa différence par rapport les méthodes de l'ingénierie dirigée par les modèles. Nous pouvons déduire que si nous choisissons soigneusement une telle méthode formelle, le raisonnement logique et l'analyse formelle resteront par conséquent les techniques et des facteurs prédominants qui influenceront et garantiront la performance des systèmes en terme, par exemple, de l'absence de défauts ou de dysfonctionnements.

Dans un premier temps nous avons expliqué la différence entre l'ingénierie dirigée par les modèles, la vérification par la simulation et la vérification par le test. Dans le premier cas, on considère la modélisation comme élément central d'une application et consiste à concevoir des applications en s'abstrayant des technologies cibles. Il propose de modéliser des applications à haut niveau d'abstraction, positionner les modèles dans les processus de conception et générer le code des applications à partir de leurs modèles. La spécification et la description des systèmes se réalisent en amont et l'implémentation se génère correctement par construction. La modélisation dans l'ingénierie dirigée par les modèles utilise dans la plupart de temps UML qui n'apporte pas de preuves formelles car c'est un langage non formel et ne permet pas d'effectuer des analyses formelles des systèmes. La méthode B est une approche basée

sur un modèle permettant le développement correct avec raffinement à partir de spécifications abstraites vers des codes exécutables, cette approche est basée sur la preuve de théorèmes et propose des méthodes de vérification de propriétés pour les systèmes produisant à partir l'ingénierie dirigée par les modèles.

Dans le deuxième et le troisième cas, on utilise ces deux techniques qui exigent plus de temps pour découvrir les bugs. Une lacune importante, dans le processus d'utilisation de la simulation et/ou de test pour la vérification et la validation, est qu'il n'y a aucun moyen de dire quand ces techniques sont terminées (lorsque tous les bugs dans le système ont été trouvés). En d'autres termes, le test et la simulation peuvent être utilisés pour démontrer la présence de bugs, mais pas leur absence.

Nous avons exposé ensuite quelques exemples de méthodes formelles servant à exprimer très rigoureusement les propriétés exigées pour un système complexe. Ces méthodes servent à prouver de manière automatisée que les propriétés souhaitées sont correctes ou non. Le Model Checking est l'une de ces méthodes. Il est un processus formel par lequel une propriété d'un comportement désiré (une spécification) est vérifiée pour un système donné (un modèle). Cette vérification se réalise en étudiant les états du système avec des techniques exhaustives. De plus, les algèbres de processus sont un formalisme mathématique pour la description et l'étude des systèmes concurrents permettant, par exemple, de vérifier les comportements indéterministes et les comportements parallèles.

Puisque nous avons besoin de modéliser en détail des applications de visualisation scientifique interactive, les réseaux de Petri semblent alors être une méthode efficace notamment grâce à leur sémantique formelle de modélisation, d'analyse et de vérification pour ces applications distribuées. Un modèle formel basé sur les réseaux de Petri se prête bien aux applications interactives si il sait vérifier leur structure en se basant sur la structure du réseau de Petri modélisant leur sémantique. La sémantique formelle souhaitée ne se base pas sur l'ingénierie dirigée par les modèles, les algèbres de processus ou le Model Checking, mais elle doit permettre à la fois d'aider l'utilisateur à construire correctement son application et vérifier sa structure en localisant des erreurs si elles existent.

Ce choix repose sur le fait que les réseaux de Petri sont à la fois une représentation mathématique et une représentation graphique explicite qui possèdent une représentation formelle avec une syntaxe et une sémantique bien définie pour fournir des preuves formelles de plusieurs propriétés, par exemple, des propriétés structurelles. En effet, ils représentent une bonne approche pour (1) modéliser des systèmes dynamiques échangeant des ressources, (2) étudier la décidabilité des propriétés et (3) intégrer des algorithmes pour les vérifier tout en gardant la classe de comportements du système étudié. Comme un outil mathématique, il est possible de mettre en place des équations d'état, des équations algébriques et d'autres modèles mathématiques dirigeant généralement le comportement de tout type de systèmes et particulièrement les différents comportement de systèmes tels que la simultanéité, la concurrence, le choix, la synchronisation, le parallélisme et le partage des ressources.

Le chapitre suivant décrit le cadre général de nos travaux concernant le modèle par

composants dédié aux applications de visualisation scientifique interactives et concernant la méthode formelle pour modéliser ces applications. Cette méthode sera utilisée pour vérifier des propriétés structurelles et pour étudier la correction de la reconfiguration dynamique des applications construites.

Deuxième partie

Contributions

LE CADRE GÉNÉRAL DES CONTRIBUTIONS

Everyone who has never made a mistake has never tried anything new.

Albert EINSTEIN

SOMMAIRE

4.1	INTRODUCTION	69
4.2	MODÈLE DE COMPOSANTS	69
4.3	FORMALISATION DU MODÈLE DE COMPOSANTS	70
4.4	RECONFIGURATION DYNAMIQUE DU MODÈLE DE COMPOSANTS	71

4.1 INTRODUCTION

Depuis plusieurs décennies, les approches à base de composants ont attiré beaucoup d'attention en génie logiciel. Plusieurs domaines typiques ont connu une réussite en exploitant cette approche comme les services Web, les systèmes critiques ou les systèmes distribués. Les modèles par composants existants sont destinés à factoriser les efforts de développement face à la complexité de construction des applications. Dans tous les modèles par composants, il existe une base commune, mais il existe aussi quelques particularités du domaine d'application visé. Cela est appliqué aux applications de visualisation scientifique interactive nécessitant une performance.

L'objectif de ce chapitre est de présenter le cadre général de nos contributions, pour le projet ANR Modèles Numériques *ExaviZ* c-à-d un modèle par composants dédié aux applications de visualisation scientifique interactives. De plus, nous présentons le cadre général de la formalisation du modèle que nous proposons en utilisant une méthode formelle à savoir les réseaux de Petri. En outre, nous présentons le principe de notre reconfiguration dynamique.

4.2 MODÈLE DE COMPOSANTS

Les scientifiques sont très intéressés à utiliser de nouveaux outils de visualisation et de rendu graphique dans le but d'améliorer la compréhension des phénomènes étudiés. En effet, les applications de visualisation scientifique interactives se basent sur

la coopération de codes d'exploration interactive des données, de codes de simulation et de supports de visualisation. Cette coopération s'appuie sur l'assemblage des différentes parties très hétérogènes de ces applications. Les modèles existants dans la littérature n'intègrent pas les contraintes de ce type d'applications comme la performance de l'application et sa reconfiguration dynamique.

Le modèle destiné à la visualisation scientifique présenté dans [138–140] se base sur des composants bloquants c-à-d ils n'itèrent que quand tous les ports d'entrée reçoivent des données. De plus, ce modèle n'a jamais été formalisé à l'aide d'une méthode précise. Notre projet de doctorat se base sur la proposition du modèle *ComSA* qui est une extension de ce modèle en enrichissant l'architecture et le comportement des composants par l'ajout de *Relations d'incidence*. Ce modèle permet de rendre un composant non bloquant au niveau de ses ports d'entrée et de ses ports de sortie c-à-d le composant n'attend pas que tous ses ports d'entrée soient alimentés pour qu'il puisse faire des traitements.

Le modèle de composants *ComSA* est basé sur une coordination exogène. S'il ne comprend pas les caractéristiques générales comme la tolérance de panne ou la découverte de services qu'on retrouve dans l'approche générale CCM présentée, section 2.4.1, il est spécifiquement conçu pour répondre aux contraintes des applications interactives à base de composants dont les fréquences sont très différentes.

Pour une telle modélisation, nous avons proposé une sémantique détaillée de chaque élément appartenant à notre modèle *ComSA* soit un composant ou un connecteur. La sémantique définie sert à vérifier qu'une modélisation est bien valide et respecte bien le comportement des composants, des connecteurs et des applications construites.

4.3 FORMALISATION DU MODÈLE DE COMPOSANTS

Après la proposition du modèle de composants *ComSA* servant à construire des applications de visualisation scientifique interactives, l'objectif suivant est de proposer une méthode pour modéliser les applications interactives développées. La méthode présentée est basée sur les réseaux de Petri et spécialement sur une classe nommée *réseaux FIFO colorés stricts*. Cette méthode sert à formaliser notre modèle par composants et décrire le comportement dynamique de nos applications. Les réseaux FIFO colorés stricts sont une sous classe des réseaux de Petri colorés comme le montre la figure 4.1.

La méthode proposée aide les utilisateurs à concevoir, construire et modifier une application à partir d'une description des composants qui la constituent, de leur assemblage et des contraintes associées. La formalisation de notre modèle permet de vérifier que les propriétés d'une application comme le démarrage, la vivacité ou l'absence d'interblocage sont bien respectées. L'étude de la vivacité des applications *ComSa* se réalise en les transformant en réseaux FIFO colorés stricts. Cette transformation aide à étudier des propriétés sur les réseaux FIFO colorés stricts comme la détection des siphons et la détection des siphons contenant des trappes. Ainsi, nous avons utilisé une

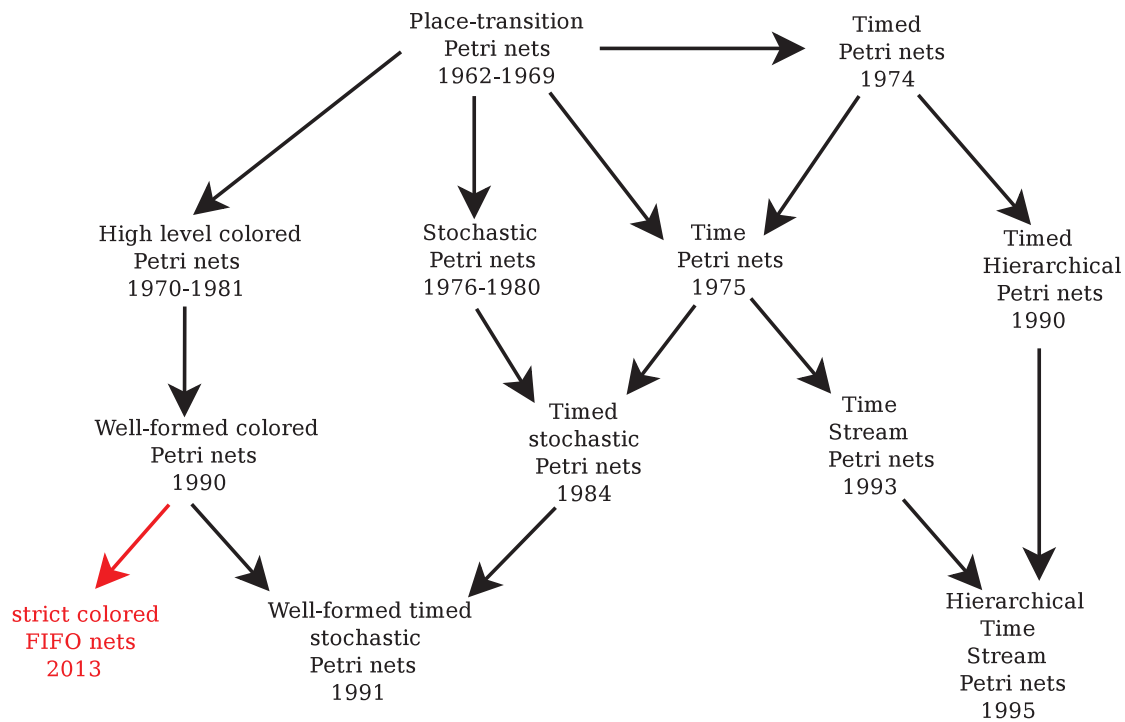


FIGURE 4.1 – Les modèles principaux de réseaux de Petri avec les réseaux FIFO colorés stricts représentés avec la couleur rouge.

traduction vers un problème de décision SAT pour l'implémentation afin d'analyser la modélisation pour détecter les éventuels blocages. L'énumération des blocages permet de définir une condition de démarrage garantissant la vivacité des applications ComSA. À titre d'exemple, la figure 4.2 montre les différentes étapes du processus d'étude de la vivacité des applications ComSA.

4.4 RECONFIGURATION DYNAMIQUE DU MODÈLE DE COMPOSANTS

Les applications à base de composants doivent faire face à des utilisateurs. La reconfiguration dynamique est devenue un besoin des utilisateurs qui souhaitent faire évoluer leur application au cours de son exécution.

Les applications visées par *ExaviZ* sont définies et composées de codes très hétérogènes qui contiennent des composants (1) de simulation pour analyser les résultats, (2) de visualisation en temps réel et (3) d'interaction avec les composants de calculs ou d'analyses. Dans les applications ComSA, nous avons étudié la reconfiguration dynamique permettant d'ajouter, d'enlever ou de remplacer un composant ou un groupe de composants. Ainsi, l'utilisateur peut, par exemple, ajouter une phase intermédiaire d'analyses.

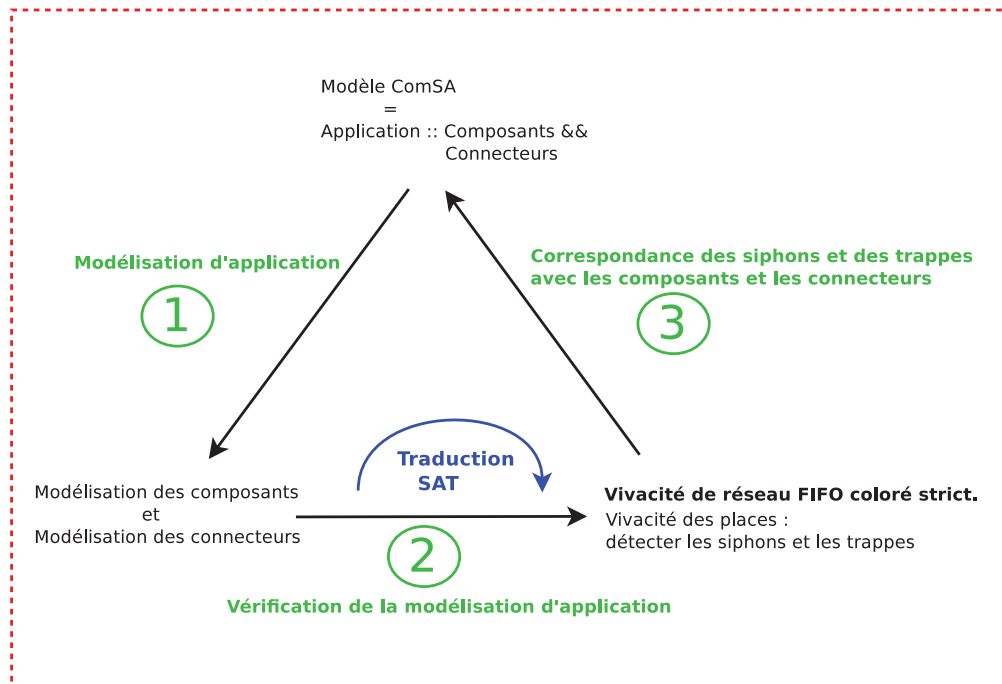


FIGURE 4.2 – Le processus d'étude de la vivacité des applications ComSA.

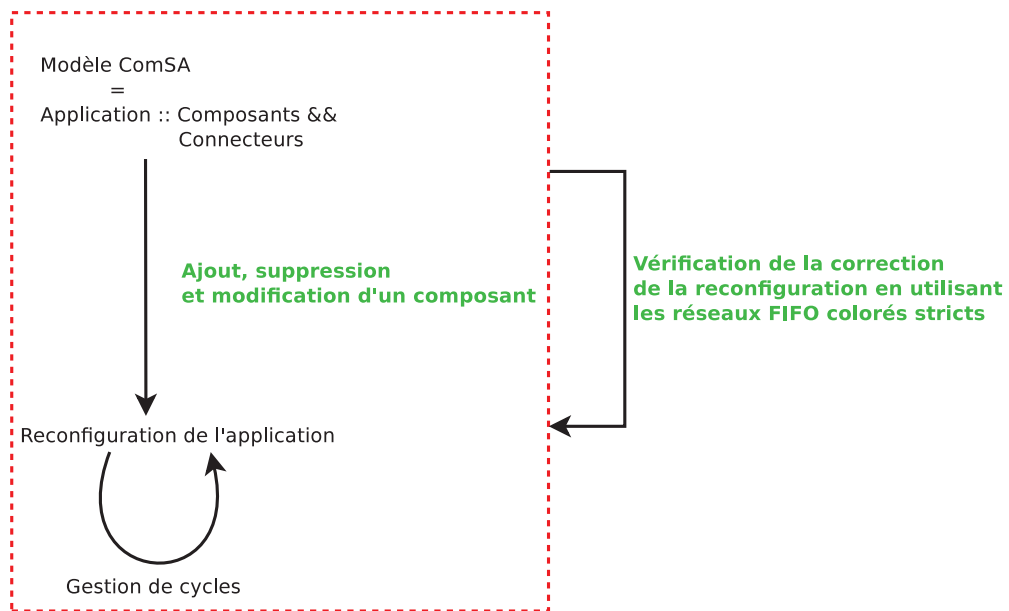


FIGURE 4.3 – Le processus de reconfiguration des applications ComSA.

Notre approche garantit une reconfiguration correcte avec le plus grand nombre possible de composants qui continueront à fonctionner au cours de la reconfiguration. Ainsi, la reconfiguration des applications ComSA permet d'arrêter juste la partie

qu'elle impacte pour minimiser les services indisponibles. Ce processus de reconfiguration est illustré par la figure 4.3.

Dans la suite de ce mémoire, Nous allons présenter un modèle par composants des applications de visualisation scientifique interactives sur lequel se base notre thèse à savoir ComSA. Nous allons introduire aussi les résultats de nos recherches concernant la formalisation de notre modèle par composants en utilisant notre classe des réseaux Petri appelé réseaux FIFO colorés stricts ainsi que l'exploitation de cette classe pour vérifier la vivacité des applications ComSA. De plus, nous allons introduire un autre résultat qui a comme objectif la reconfiguration dynamique des applications basées sur le modèle ComSA.

LE MODÈLE ComSA

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

C. A. R. HOARE

SOMMAIRE

5.1	INTRODUCTION	75
5.2	LE MODÈLE ComSA	76
5.2.1	Les composants	76
5.2.2	Les connecteurs	78
5.2.3	Les liens	80
5.3	GRAPHE D'APPLICATION	81
5.4	LA SÉMANTIQUE DU MODÈLE ComSA	83
5.5	CONCLUSION	88

5.1 INTRODUCTION

La construction et l'analyse des applications de visualisation scientifique interactive est basée sur un modèle par composants destiné à la visualisation scientifique interactive telle qu'elle est définie dans le chapitre 1. Ce modèle est basé sur une composition spatio-temporelle car il est équipé par des fonctionnalités control flow pour représenter les schémas de coordination. De plus, il est équipé par un schéma data flow pour la représentation des données. Le modèle présenté fait référence aux différents éléments constituant les applications construites et a comme objectif de coupler les différents codes de calculs, de visualisation et d'interaction.

Les applications visées sont construites à partir de composants très différents, chacun d'eux ayant ses propres comportements. Certains peuvent être lents, comme la simulation, d'autres peuvent être rapides. L'interaction introduit des contraintes de performance. En effet, le résultat doit être rapidement visible à l'utilisateur pour comprendre ce qu'il fait.

Nous présentons dans ce chapitre une approche par composants *ComSA* (Component-based approach for Scientific Applications) qui prend en compte les caractéristiques précises des applications de visualisation scientifique interactives. D'abord, ce modèle est présenté en détail avec ses différents éléments. Nous détaillons ensuite la construction des applications de visualisation scientifique interactives en se basant sur l'assemblage des éléments présentés. Finalement, ces applications vont être décrites par des graphes représentant la composition de l'application avec une formalisation de leur sémantique.

5.2 LE MODÈLE COMSA

Notre approche ComSA est basée sur des composants, des connecteurs et des liens. Les composants réalisent les différents traitements comme la simulation, le rendu graphique, l'interaction, etc. Les connecteurs et les liens permettent l'acheminement des données et l'interconnexion des composants. Les applications construites sont basées sur l'intergiciel FlowVR¹ que le modèle ComSA modélise et notamment la définition du comportement itératif des composants. De plus, les applications ComSA prennent en compte des contraintes de performance de la visualisation scientifique. Ces contraintes doivent être respectées lors de l'assemblage des composants, réalisé par des non spécialistes ou des thématiciens, pour obtenir une application fonctionnelle et performante.

5.2.1 Les composants

Un composant est une boîte noire formée de ports d'entrée et de ports de sortie qui jouent le rôle d'interfaces entre le composant et l'extérieur. Dans [140] les ports d'un tel composant sont bloquants c-à-d le composant reste en attente s'il n'a pas reçu un nouveau message sur chacun de ses ports d'entrée connectés.

Nous allons étendre la définition du composant afin de décrire les différents comportements du composant sous forme de relations d'incidence permettant de définir comment un composant consomme des données sur ses ports d'entrée et produit des résultats sur ses ports de sortie. En effet, la définition proposée ne garde plus la notion de port bloquant pour un composant mais cette notion reste valable pour une relation d'incidence donnée.

Définition 2 Un *composant* est un quadruplet :

$$C = (Id_C, pIn_C \cup \{s\}, pOut_C \cup \{e\}, IR_C).$$

- Id_C son identifiant unique ;
- pIn_C l'ensemble de ses ports d'entrée données ;
- s un port d'entrée de déclenchement ;
- $pOut_C$ l'ensemble de ses ports de sortie données avec pIn_C et $pOut_C$ disjoints ;
- e un port de sortie de signalement ;
- IR_C un ensemble de relations d'incidence ;

Un composant C possède ainsi deux types de ports : des ports d'entrée et des ports de sortie. Les ports d'entrée contiennent des ports d'entrée pIn_C qui utilisent les données produites par d'autres composants. De plus, il contient le port s de déclenchement qui mentionne à un composant qu'il peut commencer l'exécution d'une nouvelle itération. Les ports de sortie contiennent des ports de sortie $pOut_C$ qui fournissent des données aux autres composants et le port e de signalement qui indique que le composant a fini l'exécution de son itération. Les données qui circulent dans l'application

1. <http://flowvr.sourceforge.net/>

sont appelées des messages. Par contre, un composant ne connaît pas l'origine des messages qu'il reçoit ni la destination des messages qu'il produit.

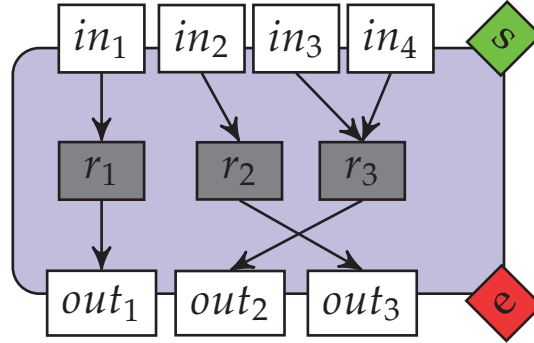


FIGURE 5.1 – Exemple d'un composant avec ses relations d'incidence.

Le comportement du composant est donc décrit par des relations d'incidence. Comme l'illustre la définition 3 elles expriment des dépendances entre les ports d'entrée et de sortie.

5.2.1.1 Relation d'incidence

Définition 3 Une relation d'incidence d'un composant C est définie par :

$$r = IR^{in} \rightarrow IR^{out}$$

où $IR^{in} \subseteq pIn_C$ est l'ensemble des ports d'entrée de r qui s'ils contiennent des données assurent que le composant produit des données sur les ports $IR^{out} \subseteq pOut_C$ de r .

Ainsi une relation d'incidence correspond à un mode opérationnel du composant qui peut être assimilé à un service précis rendu par le composant. Finalement le comportement d'un composant se définit par l'ensemble des relations d'incidence qui sont vérifiées.

Définition 4 Une relation d'incidence $r \in IR_C$ d'un composant C est vérifiée, si ses ports d'entrées contiennent des données pour qu'elle puisse produire des résultats au niveau de ses ports de sortie.

Par exemple, comme illustré par la figure 5.1, si les ports d'entrée in_3 et in_4 contiennent des données et le port s contient un signal s'il est connecté, alors la relation d'incidence r_3 est vérifiée et elle peut produire des données résultats au niveau de son port de sortie out_2 et au niveau du port e .

Pour un composant donné C , nous introduisons les notations suivantes :

- $Port_C$ désigne l'ensemble des ports d'entrée et de sortie d'un composant C ,
 $Port_C = pIn_C \cup pOut_C$;

- $IR^{in}(r)$ et $IR^{out}(r)$ désignent respectivement, l'ensemble des ports d'entrée et de sortie de la relation d'incidence r ;
- Pour un port d'entrée $p \in pIn_C$, $IR^{out}(p)$ désigne l'ensemble des relations d'incidence r tel que $p \in IR^{in}(r)$. Symétriquement, pour un port de sortie p de C , $IR^{in}(p)$ désigne l'ensemble des relations d'incidence de C tel que $p \in IR^{out}(r)$;
- Pour un ensemble de relations d'incidence \mathcal{E} , $IR^{in}(\mathcal{E})$ et $IR^{out}(\mathcal{E})$ désignent respectivement, l'ensemble des ports d'entrée et de sortie des relations d'incidence appartenant à \mathcal{E} , $IR^{in}(\mathcal{E}) = \bigcup_{r \in \mathcal{E}} IR^{in}(r)$ et $IR^{out}(\mathcal{E}) = \bigcup_{r \in \mathcal{E}} IR^{out}(r)$;
- $IR^{out}(pIn_C)$ désigne l'ensemble des ports de sortie des relations d'incidence vérifiées par l'ensemble pIn_C c-à-d $IR^{out}(pIn_C) = \{o \in IR^{out}(r) | IR^{in}(r) \subseteq pIn_C\}$.

5.2.1.2 Comportement d'un composant

Dans le cadre des applications visées, les composants sont itératifs et ne connaissent pas la source des données reçues et la destination des données envoyées. La seule connaissance de chaque composant est la liste de ses ports d'entrée, de ses ports de sortie et de ses relations d'incidence. Pour notre modèle, le processus itératif d'un composant est une boucle appelée « *wait-get-put* » illustrée figure 5.2 et définie par :

Définition 5 *L'itération d'un composant appelé « wait-get-put » consiste à :*

wait : Attendre

- *des données sur tous les ports d'au moins une relation d'incidence du composant ;*
- *un signal de déclenchement sur son port s s'il est connecté.*

get : Consommer une donnée sur tous les ports d'entrée des relations d'incidence vérifiées pour exécuter la tâche correspondante à chacune des relations vérifiées.

put : Envoyer

- *les résultats produits sur les ports de sortie de toutes les relations d'incidence vérifiées ;*
- *sur le port e un signal notifiant la fin d'une itération.*

Chaque composant gère en interne un numéro d'itération qui permet de suivre ce processus « *wait-get-put* ». Ce numéro d'itération est également indiqué en entête de chaque message produit par le composant sur les ports de sortie des relations d'incidence vérifiées.

5.2.2 Les connecteurs

Assembler des composants, pour réaliser une application, consiste à définir un schéma de communications permettant les échanges de données sur les interfaces des composants soient leurs ports d'entrée et de sortie. Ce schéma de communications se base sur des connecteurs et des liens. Les connecteurs réalisent l'acheminement des

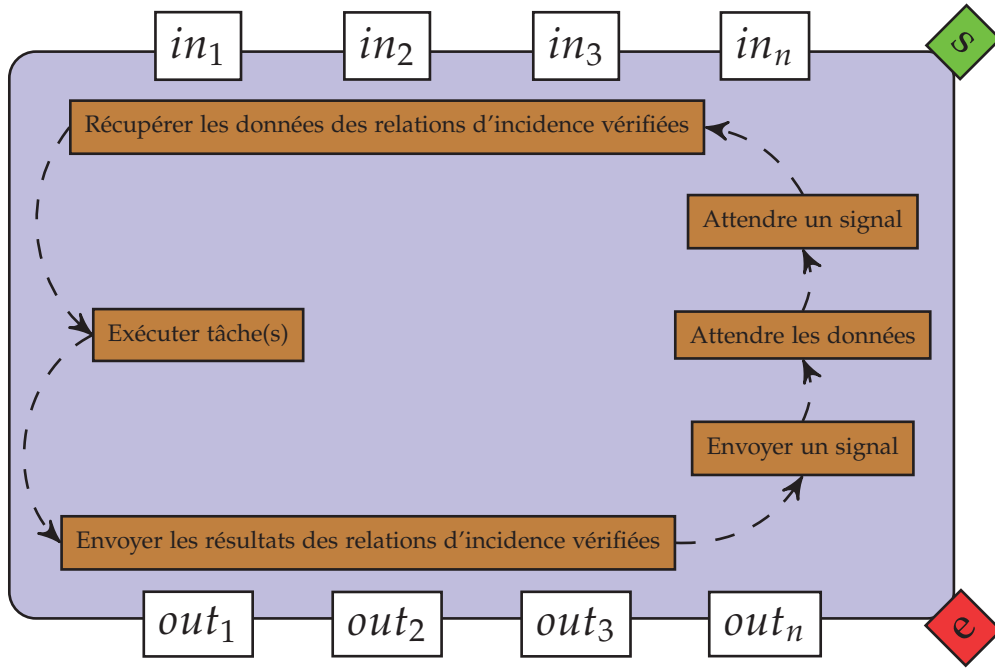


FIGURE 5.2 – Le comportement du composant du modèle ComSA.

messages selon une politique adaptée à la communication entre des composants itératifs ayant des fréquences différentes. Les liens formalisent les liaisons entre composants et connecteurs. Dans la littérature, il existe trois classes de connecteurs chacune avec son comportement bien défini [22, 164] :

- **communication synchrone** permettant d'échanger les données directement. Dans cette communication le composant émetteur et le composant récepteur sont cadencés à la même horloge. Le composant récepteur reçoit de façon continue les informations au rythme où le composant émetteur les envoie ;
- **communication asynchrone** peut avoir un buffer limité. Ce buffer est utilisé pour stocker les données envoyées par le composant émetteur, mais pas encore distribuées vers le composant récepteur ;
- **communication avec perte** livre seulement certaines données reçues et perd le reste.

Dans notre modèle ComSA, nous avons choisi des connecteurs qui appartiennent aux classes des communications définies ci-dessus.

Définition 6 Un connecteur est un quadruplet

$$con = (Id, \{i, s\}, \{o\}, t)$$

où i est un port d'entrée, o un port de sortie et t son type. Les paramètres Id et s sont similaires à leurs homonymes dans un composant.

Dans le cadre des applications de visualisation scientifique interactives, certains composants, par exemple, de la partie visualisation, ont besoin de fonctionner à leur propre fréquence. Dans ce cas, il faut offrir des schémas de communication qui permettent de fournir des messages vides, par exemple, quand un composant récepteur est prêt et que le composant émetteur n'a pas encore envoyé une nouvelle donnée.

Dans ComSA, nous avons défini trois types de connecteurs [140] qui couvrent les cas d'utilisations les plus connus. Ces connecteurs sont illustrés figure 5.3 et sont les suivants :

- **sFIFO** est une liaison simple de type file FIFO où, pour éviter les débordements, l'émetteur attend un signal de déclenchement sur son port *s* en général envoyé par le récepteur. Les composants récepteurs peuvent ralentir les composants émetteurs et réciproquement. Cela influence sur la fréquence globale d'une application ce qui peut impliquer un impact négatif sur son interactivité ;
- **bBuffer** et **nbBuffer** conservent les messages et n'en délivrent un que s'ils ont reçu un signal de déclenchement sur leur port *s*. Le **nbBuffer** est la variante non bloquante du **bBuffer** c-à-d il génère un message vide pour le récepteur lorsqu'il est déclenché alors qu'il n'a aucun message en attente ;
- **bGreedy** et **nbGreedy** ne stockent que le dernier message reçu et le délivrent à la réception d'un signal de déclenchement sur leur port *s*. Le **nbGreedy** est la variante non bloquante du **bGreedy**.

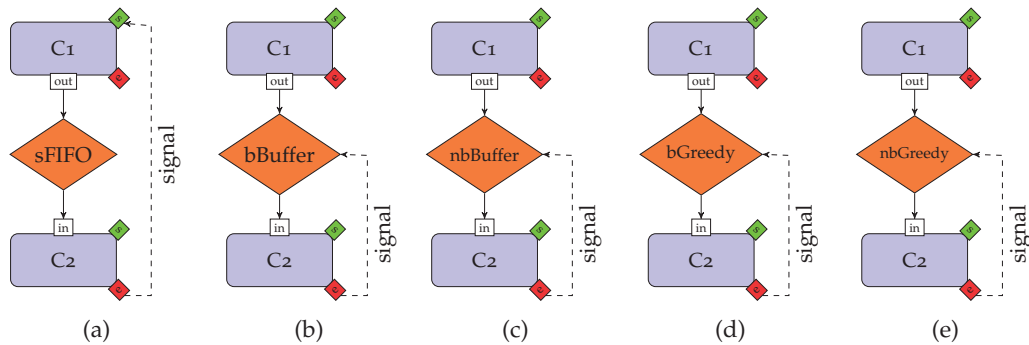


FIGURE 5.3 – Les connecteurs : (a) sFIFO, (b) bBuffer, (c) nbBuffer, (d) bGreedy et (e) nbGreedy.

5.2.3 Les liens

Les liens permettent de relier les composants et les connecteurs via leurs ports.

Définition 7 Un lien est un couple $\langle x^p, y^q \rangle$ où x, y sont des composants ou des connecteurs, $p \in pOut_x \cup \{e\}$ et $q \in pIn_y \cup \{s\}$. Lorsque $p \neq e$, $q \neq s$ et que x ou y est un connecteur le lien est dit lien de données. Lorsque $p = e$, $q = s$ et que x est un composant alors le lien est dit lien déclencheur.

Définition 8 Soient $\langle x_1^{p_1}, y_1^{q_1} \rangle$ et $\langle x_2^{p_2}, y_2^{q_2} \rangle$ deux liens de données. $\langle x_1^{p_1}, y_1^{q_1} \rangle$ et $\langle x_2^{p_2}, y_2^{q_2} \rangle$ sont dit compatibles si $y_1^{q_1} \neq y_2^{q_2}$.

Il existe deux types de liens :

- **les liens de données** : ces liens transmettent les données des ports de données entre les connecteurs et les composants. Un connecteur est toujours positionné entre deux composants pour définir la politique de communication entre les composants. Un lien de données n'est connecté qu'avec un seul port d'entrée ou de sortie d'un connecteur ou d'un composant ;
- **les liens de déclenchement** : les signaux déclencheurs sont transmis à l'aide des liens de déclenchement. Ces liens représentent le control flow dans le modèle ComSA. D'autre part, si un port s d'un composant est connecté avec plusieurs liens de déclenchement, le composant ne commence pas son itération s'il n'a pas reçu un signal de chacun de ces liens connectés.

5.3 GRAPHE D'APPLICATION

Finalement, l'application va être décrite par un graphe qui connecte les ports de sortie aux ports d'entrée des composants et des connecteurs. De plus, un port d'entrée de données ne peut être alimenté que par un seul lien.

Pour la suite nous considérons les définitions suivantes :

Définition 9 Une application *App* est définie par le graphe $(Comp \cup Conn, DI \cup TI)$ où *Comp* est l'ensemble des composants, *Conn* l'ensemble des connecteurs, *DI* l'ensemble des liens de données deux à deux compatibles et *TI* les liens déclencheurs.

Définition 10 Dans une application $(Comp \cup Conn, DI \cup TI)$, le port p d'un composant x est dit connecté s'il est le sommet d'un des arcs de $DI \cup TI$.

Définition 11 Dans une application $(Comp \cup Conn, DI \cup TI)$, un canal de communication entre un port de sortie o d'un composant C_1 et un port d'entrée i d'un composant C_2 passe toujours à travers un connecteur c et est décrit par les deux liaisons de données $\langle o, c^i \rangle, \langle c^o, i \rangle$. Ce canal est défini par $C_1^o \xrightarrow{c} C_2^i$.

Dans une application, les composants doivent être connectés de façon à assurer que les données arrivant sur un port d'entrée pourront être consommées par au moins une relation d'incidence d'une part et qu'un port de sortie connecté fournira bien des données d'autre part. La définition suivante formalise ces notions.

Définition 12 Notons pIn_C^c (resp. $pOut_C^c$) l'ensemble des ports d'entrée (resp. de sortie) connectés pour un composant C dans une application *App*. On dit alors qu'une relation d'incidence r est déclenchable si $IR^{in}(r) \subseteq pIn_C^c$. Notons $IR_C^c \subseteq IR_C$ l'ensemble des relations d'incidence qui peuvent être déclenchées à partir de pIn_C^c alors :

- un composant est bien connecté en entrée si $IR^{in}(\mathcal{IR}_C^c) = pIn_C^c$;
- un composant est bien connecté en sortie si $IR^{out}(\mathcal{IR}_C^c) \subseteq pOut_C^c$;
- une application *App* est bien formée si tous ses composants sont bien connectés en entrée et en sortie.

Une application *bien formée* permet de prévenir les cas de saturation des ports d'entrée de relations d'incidence qui ne sont jamais déclenchées et les cas de blocage dus à la connexion d'un port de sortie de ces mêmes relations d'incidence.

L'application illustrée par la figure 5.4, s'inspire du projet *ExaviZ* et est basée sur une simulation de dynamique moléculaire. Dans cette application, le composant *Simulation* génère des positions d'atomes affichés en temps réel par le composant *Visualisation*. De plus, un composant *Interaction* (par exemple gérant un Omni Phantom[®]) permet de contrôler la position 3D d'un avatar évoluant parmi les atomes. Il permet aussi d'activer une force calculée par un composant *ForceGenerator* à partir d'une distance entre l'avatar et l'atome qui a été sélectionné, force renvoyée à la simulation pour influencer la dynamique moléculaire. Lorsque la force est appliquée, le composant *Simulation* génère également l'énergie du système d'atomes, énergie qui peut être affichée par *Visualisation*.

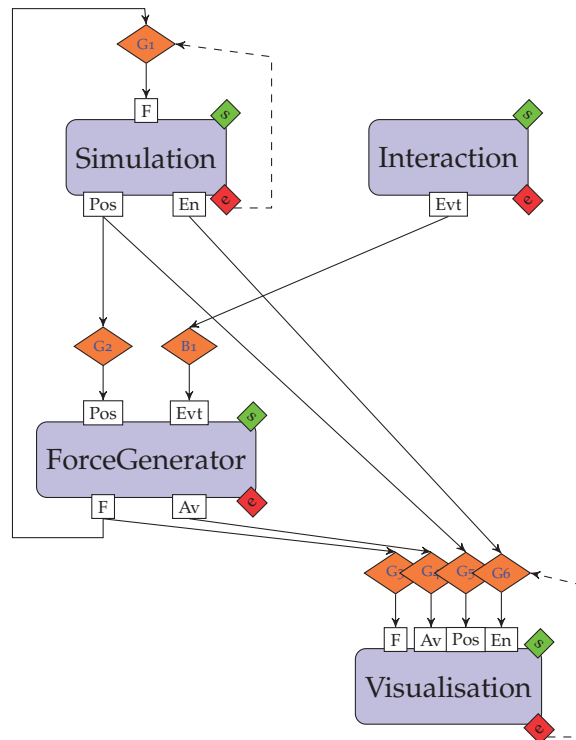


FIGURE 5.4 – Exemple de graphe d'application (seuls deux liens déclencheurs ont été représentés pour ne pas charger le graphe).

D'après les définitions 2 et 3, les composants sont définis comme suit :

- $C_S = \{ \text{"Simulation"}, F, s, \{Pos, En, e\}, IR_S \}$
avec $IR_S = \{ \emptyset \rightarrow \{Pos\}, \{F\} \rightarrow \{Pos, En\} \}$;
- $C_I = \{ \text{"Interaction"}, s, \{Evt, e\}, IR_I \}$
avec $IR_I = \{ \emptyset \rightarrow \{Evt\} \}$;
- $C_F = \{ \text{"ForceGenerator"}, \{Pos, Evt, s\}, \{Av, F, e\}, IR_F \}$
avec $IR_F = \{ \{Pos, Evt\} \rightarrow \{Av, F\}, \{Evt\} \rightarrow \{Av\} \}$;
- $C_V = \{ \text{"Visualisation"}, \{F, Av, Pos, En, s\}, \{e\}, IR_V \}$
avec $IR_V = \{ \{F\} \rightarrow \emptyset, \{Av\} \rightarrow \emptyset, \{Pos\} \rightarrow \emptyset, \{En\} \rightarrow \emptyset \}$.

Afin de garantir la performance, le schéma de communications est basé sur des connecteurs de type *Greedy* afin de perdre les données qui pourraient saturer les composants les plus lents. Cependant, les événements émis par *Interaction* qui déclenchent le calcul de la force influençant le cours de la simulation sont de type clic souris et ne peuvent donc pas être perdus. D'où un connecteur de type *Buffer* entre *Interaction* et *ForceGenerator*. Le choix entre des connecteurs de type bloquant ou non est fortement lié aux relations d'incidence des composants. Par exemple, les relations d'incidence de *ForceGenerator* montrent que le port *Evt* peut être qualifié de bloquant car le composant ne procède à une itération que si ce port est alimenté. Donc, en fonction de la capacité de *ForceGenerator* à traiter des messages vides, nous choisirons un *bBuffer* ou un *nbBuffer*, le second permettant de préserver la performance du module. En revanche, le port *Pos* n'étant pas dans les deux relations d'incidence, est dit non bloquant et le connecteur *G2* peut être un *bGreedy* sans que cela influence la fréquence de *ForceGenerator*. Il en est de même pour les composants *Simulation* et *Visualisation* qui n'ont aucun port bloquant. Les connecteurs en amont peuvent être simplement des connecteurs *bGreedy* qui ont l'avantage d'être moins coûteux en ne générant pas de messages vides.

Le port *s*, même si ce n'est pas utilisé sur cet exemple, est toujours un port bloquant et permet de contrôler, s'il est connecté, les itérations du composant quel que soit ses relations d'incidence.

5.4 LA SÉMANTIQUE DU MODÈLE COMSA

Pour une modélisation des comportements de notre modèle ComSA, il est donc nécessaire de formaliser la sémantique des composants, des connecteurs et d'une application afin ensuite de démontrer qu'une modélisation est bien conforme à cette sémantique.

Pour un composant et un connecteur il s'agit de formaliser l'opération effectuée lorsqu'ils sont déclenchés. Pour cela nous devons introduire des définitions supplémentaires, en particulier pour exprimer la consommation et la production des messages. Dans ces définitions, les composants et les connecteurs se déclenchent uniquement en fonction de l'état de leurs ports d'entrée, leur comportement indique donc comment le composant ou connecteur met à jour son état quand ils sont déclenchés. L'application, par l'intermédiaire des liens, va elle aussi mettre à jour l'état de ces objets en vidant les ports de sortie des uns pour remplir les ports d'entrée des autres.

Nous aurons besoin de la notion de file d'attente (FIFO) avec les opérations classiques suivantes : si f est une file FIFO et e un élément, on note $f+e$ l'ajout de e dans la file FIFO, $f \uparrow$ le prochain élément à sortir de la file FIFO, c-à-d le plus ancien, cette fonction est définie uniquement si f est non vide. $f--$ est la suppression de l'élément le plus ancien de la file. Enfin \emptyset représente la file FIFO vide.

Définition 13 Soit $\vec{P} = (p_i)_{1 \leq i \leq n}$ un vecteur de ports distincts et $\vec{M} = (m_i)_{1 \leq i \leq m}$ un vecteur de messages. On dit que \vec{M} et \vec{P} sont compatibles s'ils ont la même longueur et s'ils sont ordonnés de telle manière que le message m_i est destiné au port p_i . Le message m_i peut être noté vide $m_i = \emptyset$ pour signifier l'absence de message.

Pour un vecteur de messages \vec{M} compatible avec \vec{P} , la notation m_{p_i} (où $p_i \in \vec{P}$) désigne le message m_i de \vec{M} .

Définition 14 Soit \vec{pIn}_C le vecteur des ports d'entrée d'un composant C et \vec{M} un vecteur de messages compatible, on dit que \vec{M} vérifie $r \in IR_C$ si et seulement si $\forall p \in IR^{in}(r) m_p \neq \emptyset$. On note $V_C(\vec{M})$ l'ensemble des relations d'incidence de C vérifiées par \vec{M} .

Un composant, selon notre modèle ComSA, procède à une itération dès que ses ports sont alimentés pour au moins une de ses relations d'incidence. Les messages correspondants sont alors consommés et les autres messages qui peuvent ne pas être vides attendent la prochaine itération. Enfin les résultats produits sont portés sur les ports de sortie des relations d'incidence ayant déclenchées l'itération. Soient Φ_C et Ψ_C les deux fonctions ci-dessous qui permettent de définir la consommation et la production des messages lorsqu'un composant procède à son itération.

Définition 15 Soit Φ_C la fonction qui, à partir d'un vecteur de messages en entrée d'un composant compatible avec les ports d'entrée, construit le vecteur de messages restant en entrée à la fin de l'itération.

$$\Phi_C(\vec{M}) = \vec{M}' \text{ tel que } m'_p = \begin{cases} \emptyset & \forall p \in IR^{in}(V_C(\vec{M})) \\ m_p & \text{sinon} \end{cases}$$

Définition 16 Soit Ψ_C la fonction qui, à partir d'un vecteur de messages \vec{M} en entrée d'un composant et compatible avec les ports d'entrée, construit le vecteur de messages \vec{M}' portés sur les ports de sortie.

$$\Psi_C(\vec{M}) = \vec{M}' \text{ tel que } \vec{M}' \text{ est compatible avec } \vec{pOut}_C \text{ et } m'_p = \begin{cases} \omega_p & \text{si } p \in \bigcup_{r \in V_C(\vec{M})} IR^{out}(r) \\ \emptyset & \text{sinon} \end{cases}$$

où ω_p est le résultat produit par C à destination du port p .

Soient \vec{M}^{In} un vecteur de messages compatible avec \vec{pIn}_C , \vec{M}^{Out} un vecteur de messages compatible avec \vec{pOut}_C , it le numéro d'itération du processus itératif du

composant et m_e, m_s les messages associés aux ports de signalement et de déclenchement, l'état d'un composant est défini par le tuple $(\vec{M}^{In}, \vec{M}^{Out}, it, m_e, m_s)$. L'itération d'un composant est alors formalisée par le passage de l'état $(\vec{M}^{In}, \vec{M}^{Out}, it, \emptyset, 1)$ à l'état $(\Phi(\vec{M}^{In}), \Psi(\vec{M}^{In}), it + 1, 1, \emptyset)$ lorsque $V_C(\vec{M}^{In}) \neq \emptyset$ (\vec{M}^{Out} désigne un vecteur de messages vides).

Si le port s d'un composant n'est pas connecté dans l'application, on considère que m_s est remis à 1 après chaque itération. Le déclenchement du composant ne dépend alors que de la vérification d'au moins une relation d'incidence.

La sémantique des connecteurs est plus simple à définir. Le connecteur *sFIFO* peut s'assimiler à un simple fil qui laisse transiter les messages sachant qu'il ne peut recevoir qu'un seul message à la fois. Ainsi si l'état du connecteur est décrit par (m_{in}, m_{out}) où m_{in} est le message sur son port i et m_{out} est le message porté sur son port o , ce connecteur passe de l'état (m_{in}, \emptyset) à l'état (\emptyset, m_{in}) .

Pour les deux autres familles de connecteurs, il faut définir leur manière de gérer les messages qu'ils reçoivent avec stockage pour les connecteurs de type *Buffer* ou avec perte pour les connecteurs de type *Greedy*.

L'état d'un connecteur de type *Greedy* est décrit par un tuple $(m_{in}, m_{out}, m_{store}, m_s)$ où m_{in} est le message sur son port i , m_{out} le message porté sur o , m_{store} le message conservé par le connecteur et m_s le message associé au port s du connecteur. Le déclenchement sera différent suivant que le connecteur est bloquant ou non. Le message vide (qui est différent de l'absence de message) est noté m_\emptyset . Les comportements sont résumés par la table 5.1. Dans cette table les messages en gras indiquent la présence obligatoire d'une donnée.

bGreedy	
État de déclenchement	État final
$(m_{in}, \emptyset, \mathbf{m}_{store}, 1)$	$(m_{in}, \mathbf{m}_{store}, \emptyset, \emptyset)$
$(\mathbf{m}_{in}, \emptyset, m_{out}, \emptyset)$	$(\emptyset, \emptyset, \mathbf{m}_{in}, \emptyset)$
nbGreedy	
État de déclenchement	État final
$(m_{in}, \emptyset, \mathbf{m}_{store}, 1)$	$(m_{in}, \mathbf{m}_{store}, \emptyset, \emptyset)$
$(\mathbf{m}_{in}, \emptyset, \emptyset, 1)$	$(\emptyset, m_\emptyset, \emptyset, \emptyset)$
$(\mathbf{m}_{in}, \emptyset, m_{store}, \emptyset)$	$(\emptyset, \emptyset, \mathbf{m}_{in}, \emptyset)$

TABLE 5.1 – Les états des connecteurs de type *Greedy*.

Pour les connecteurs de type *Buffer* ce n'est plus le dernier message qui est stocké mais tous les messages qui arrivent sur le port i . Ainsi l'état du connecteur est représenté par le tuple $(m_{in}, m_{out}, L_{store}, m_s)$ où m_{in} est le message sur son port i , m_{out} le message porté sur o , L_{store} la file FIFO des messages contenus dans le buffer de ces

connecteurs et m_s le message porté sur le port s . Alors, la table 5.2 synthétise le nouvel état du connecteur lorsqu'il est déclenché. Ces connecteurs sont déclenchés soit à l'arrivée d'un signal $m_s = 1$ sur le port s soit à l'arrivée d'un message sur i . Comme dans la table 5.2, les messages écrits en gras indiquent la présence effective d'une donnée. Cette convention est étendue pour la file FIFO L_{store} .

bBuffer	
État de déclenchement	État final
$(m_{in}, \emptyset, \mathbf{L_{store}}, 1)$	$(m_{in}, \mathbf{L_{store}}\uparrow, \mathbf{L_{store}}\--, \emptyset)$
$(\mathbf{m_{in}}, \emptyset, L_{store}, \emptyset)$	$(\emptyset, \emptyset, L_{store} + \mathbf{m_{in}}, \emptyset)$
nbBuffer	
État de déclenchement	État final
$(m_{in}, \emptyset, \mathbf{L_{store}}, 1)$	$(m_{in}, \mathbf{L_{store}}\uparrow, \mathbf{L_{store}}\--, \emptyset)$
$(m_{in}, \emptyset, \emptyset, 1)$	$(m_{in}, m_{\emptyset}, \emptyset, \emptyset)$
$(\mathbf{m_{in}}, \emptyset, L_{store}, \emptyset)$	$(\emptyset, \emptyset, L_{store} + \mathbf{m_{in}}, \emptyset)$

TABLE 5.2 – Les états des connecteurs de type Buffer.

La sémantique d'une application peut être définie de proche en proche grâce à la sémantique des composants et des connecteurs. La construction d'une application consiste à assembler les composants et les connecteurs par des liens. Ces liens représentent de simples arêtes dans le graphe et peuvent être considérés comme des fils laissant passer les messages les uns après les autres. Ils agissent donc comme des files FIFO.

Pour définir l'état global d'une application App dont le graphe est $(Comp \cup Conn, DI \cup TI)$, on associe à chaque lien $l \in DI \cup TI$ la file d'attente \vec{l} . L'état de App est donc l'union des états de chaque connecteur, de chaque composant et des états de chaque file d'attente associée aux liens.

Ainsi, lorsque l'application est dans un état donné, elle va passer dans l'état suivant en appliquant une des règles de déclenchement d'un des objets de notre modèle, connecteur, composant ou lien. Ces règles de déclenchement modifient l'état courant de l'objet déclenché mais également l'état d'autres objets de la manière suivante :

- A la fin d'une étape d'un connecteur c , un message porté sur un port de sortie va alimenter les files d'attente de tous les liens $\langle c^o, C^i \rangle$ avec o le port de sortie du c , $C \in Comp$ et $i \in pIn_C$;
- A la fin d'une étape d'un composant C , un message porté sur un port de sortie va alimenter les files d'attente de tous les liens $\langle C^o, c^i \rangle$ avec $o \in pOut_C$, $c \in Conn$ et $i \in pIn_c$;
- Suite à la mise à jour de la file d'attente d'un lien de données $\langle C^o, c^i \rangle$ $C \in$

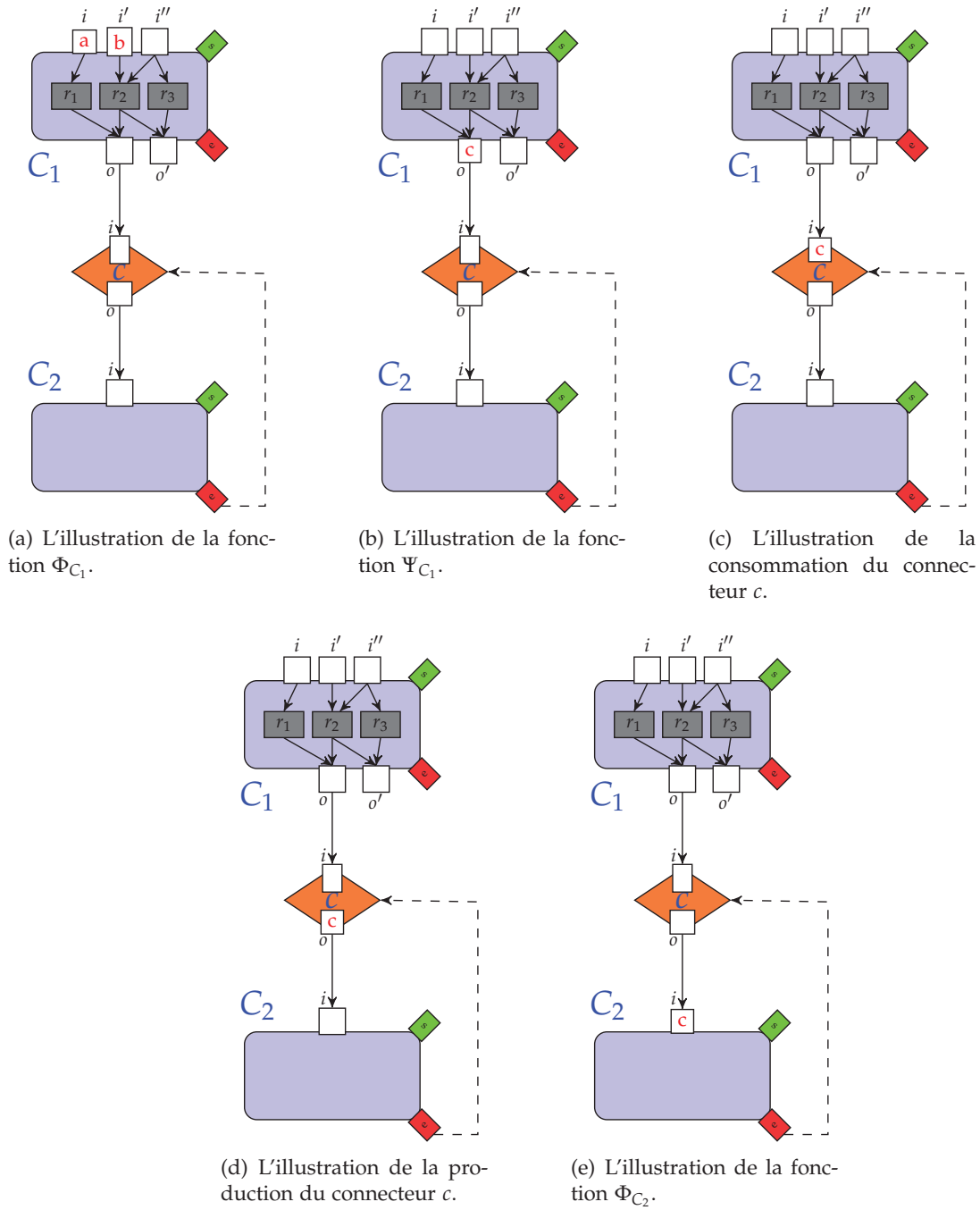


FIGURE 5.5 – L'illustration de la sémantique des composants et des connecteurs ComSA.

$Comp$, $o \in pOut_C$, $c \in Conn$ et $i \in pIn_c$. C va être mis à jour puisque $\overrightarrow{M^{In}}$ est modifié;

- Suite à la mise à jour de la file d'attente d'un lien de données $\langle c^o, C^i \rangle$ $c \in Conn$, o est le port de sortie du connecteur c c-à-d $o \in pOut_c$, $C \in Comp$ et $i \in pIn_C$. c va être mis à jour puisque m_{in} est modifié ;
- Le message $m_s = 1$ est porté sur un port de déclenchement s d'un connecteur ou d'un composant C , l'état de C est alors modifié, lorsque tous les liens $\langle c^e, C^s \rangle \in Tl$ avec $C \in Comp$ ont des files d'attente qui contiennent au moins un signal ;
- A la fin du déclenchement d'un connecteur c , l'état de tous les liens $\langle c^o, C^i \rangle$ où o est le port de sortie du connecteur c et $C \in Comp$ avec $i \in pIn_C$ est mis à jour ;
- A la fin du déclenchement d'un composant C , l'état de tous les liens $\langle C^o, c^i \rangle$ où $o \in pOut_C$ et $c \in Conn$ avec $i \in pIn_c$ est mis à jour ;
- A la fin du déclenchement d'un composant C , l'état de tous les liens $\langle C^e, c^s \rangle$ où $c \in Conn$ sont mis à jour.

La figure 5.5 illustre les différentes fonctions de la consommation et la production des messages de deux composants C_1 et C_2 connectés à l'aide d'un connecteur c du type *bBuffer*. La figure 5.5(a) montre qu'au niveau du composant C_1 , le port i contient le message a et le port i' contient le message b . En effet, l'état actuel du composant C_1 est $(\vec{M}^{In}, \vec{M}^{Out}, it, m_e, m_s)$ avec : $\vec{M}^{In} = (a, b, \emptyset)$, $\vec{M}^{Out} = \vec{M}_{\emptyset}$, $it = x$, $m_e = 1$ et $m_s = 0$. Puisque $V_C(\vec{M}^{In}) \neq \emptyset$ avec $V_C(\vec{M}^{In}) = (r_1, r_2)$, alors les fonctions $\Phi(\vec{M}^{In})$ et $\Psi(\vec{M}^{In})$ peuvent être appliquées. Nous obtenons l'état suivant, illustré figure 5.5(b), formalisé par $(\Phi(\vec{M}^{In}), \Psi(\vec{M}^{In}), it, m_e, m_s)$ avec $\Phi(\vec{M}^{In}) = \vec{M}_{\emptyset}$, $\Psi(\vec{M}^{In}) = (c, \emptyset)$, $it = x + 1$, $m_e = 0$ et $m_s = 1$. Cet état permet de produire le message c sur le port de sortie o et un signal de fin d'itération sur le port e du composant C_1 .

Par la suite, le composant C_1 va alimenter les files d'attente de tous les liens connectés avec le port o . En effet, C_1 va être mis à jour et son état est le suivant : $(\vec{M}_{\emptyset}, \vec{M}_{\emptyset}, it, 1, 0)$. De plus, l'état $(m_{in}, m_{out}, L_{store}, m_s)$ du connecteur c est modifié avec $m_{in} = c$, $m_{out} = \emptyset$, $L_{store} = \emptyset$ et $m_s = 0$, illustré figure 5.5(c). Le connecteur c va modifier son état tout seul puisqu'il a reçu une donnée sur son port d'entrée i , alors son état est : $(\emptyset, \emptyset, L_{store}+c, 0)$.

Quand le composant C_2 envoie un signal vers le connecteur c pour le déclencher, l'état du connecteur c modifié est $(\emptyset, \emptyset, L_{store}, 1)$ avec L_{store} contient juste le message c . De plus, le connecteur va produire une donnée sur son port de sortie o , donc son état est : $(\emptyset, c, L_{store}-, 0)$, illustré figure 5.5(d). Après cela, le composant C_2 change son état $(\vec{M}^{In}, \vec{M}^{Out}, it, m_e, m_s)$ avec : $\vec{M}^{In} = (c)$, $\vec{M}^{Out} = \vec{M}_{\emptyset}$, $it = x$, $m_e = 1$ et $m_s = 0$, illustré figure 5.5(e).

5.5 CONCLUSION

Dans ce chapitre, nous avons introduit le modèle de composant itératif ComSA permettant de modéliser des applications de visualisation scientifique interactives. Nous avons présenté les différents concepts relatifs à ComSA. Ensuite, nous avons exposé la sémantique de notre modèle sur laquelle se base cette thèse.

Le modèle de composant ComSA peut être mis en œuvre dans des intergiciels comme FlowVR [16] qui est spécifiquement conçu pour assurer la coordination exogène qui répond aux contraintes et aux besoins des applications de visualisation scientifiques interactive. Construire une application ComSA peut être une tâche difficile, surtout pour les utilisateurs qui ne sont pas des experts en informatique. En outre, la construction de ces applications peut contenir des blocages qui empêchent l'exécution de l'application. En particulier, la composition doit être bien définie afin d'assurer que l'application puisse commencer et aussi pour éviter les problèmes à l'exécution. La résolution de ces problèmes est souvent basée sur l'analyse de blocages de l'application. Pour les approches à base de composants, les blocages sont largement étudiés comme illustré par ce travail récent [27]. Dans ce contexte, notre objectif est de fournir un ensemble d'outils qui permettent de construire l'application et de l'analyser. En particulier, des outils d'analyse visant à corriger l'application par la détection de blocages. Ils visent également à démarrer correctement l'application ce qui peut être non trivial.

Les réseaux de Petri sont largement utilisés comme modèles formels pour les systèmes concurrents. Pour les approches à base de composants, ils sont aussi fréquemment utilisés en raison de leur capacité d'analyse et de vérification comme dans [35, 111, 218] où l'analyse du comportement de l'application et le raffinement sont basés sur les réseaux de Petri. En outre, ils offrent une représentation graphique simple qui peut être utilisée à des fins de simulation.

Fournir une sémantique formelle qui tienne compte des contraintes des applications ComSA comme la fiabilité et la performance permettrait de fournir une vérification efficace de certaines propriétés des applications telles que la vivacité. Dans le chapitre suivant, nous présentons cette sémantique formelle basée sur le formalisme des réseaux de Petri. De plus, l'analyse et la détection de blocages dans les applications ComSA vont être étudiés.

LA MODÉLISATION DU MODÈLE ComSA

Management by objective works - if you know the objectives. Ninety percent of the time you don't.

Peter F. DRUCKER

SOMMAIRE

6.1	INTRODUCTION	91
6.2	MODÉLISATION EN RÉSEAUX FIFO COLORÉS	92
6.2.1	Réseaux FIFO Colorés stricts	92
6.2.2	Modèle des composants en sCFN	98
6.2.3	Modèle des connecteurs en sCFN	103
6.2.4	Modèle d'une application en sCFN	106
6.3	L'ANALYSE DES BLOCAGES DES APPLICATIONS ComSA	109
6.3.1	La vivacité des sCFN	110
6.3.2	La configuration de démarrage des applications ComSA	119
6.4	CONCLUSION	122

6.1 INTRODUCTION

Le modèle ComSA vise à faciliter la spécification et la construction des applications de visualisation scientifique interactive. Ces applications sont composées de codes très hétérogènes comportant des composants : (1) de simulation dont on souhaite analyser les résultats, (2) d'analyse qui peuvent être lancés ponctuellement, (3) de visualisation temps réel et (4) d'interaction pour interagir avec la simulation ou les étapes d'analyse. Dans le cadre de notre travail, la description de la sémantique est insuffisante, par exemple, pour s'assurer de bon fonctionnement en terme de vivacité. En plus de la sémantique décrivant les différents éléments construisant nos applications, nous avons besoin de connaître comment les composants se comportent pour pouvoir analyser si notre application est correcte. En particulier, on cherche à vérifier et valider les propriétés de vivacité et de blocage. Pour cet objectif, le travail proposé dans cette thèse s'appuie sur les réseaux de Petri.

Dans ce chapitre, nous présentons le modèle formel que nous utilisons et qui repose sur le formalisme des réseaux de Petri. Le but est de pouvoir spécifier et modéliser formellement le comportement des applications ComSA. En particulier, dans le

contexte de cette thèse, les applications ComSA sont décrites par une classe des réseaux de Petri appelée réseaux FIFO colorés stricts (strict Colored FIFO Nets, sCFN). Cette classe peut capturer précisément la sémantique du comportement de notre modèle et est utilisée pour vérifier nos propriétés.

Nous commençons ce chapitre par introduire intuitivement le modèle formel que nous proposons pour modéliser les différents comportements des composants et les différentes politiques de communication au sein d’une application ComSA. Puis, nous utilisons notre modèle formel pour modéliser les applications ComSA construites. Nous présentons ensuite l’étude de démarrage et de vivacité des applications ComSA.

6.2 MODÉLISATION EN RÉSEAUX FIFO COLORÉS

Dans cette section nous allons montrer comment modéliser notre modèle de composants en réseaux FIFO colorés. Les réseaux que nous allons définir sont un peu différents des réseaux de Petri colorés introduits dans la littérature car dans notre définition toutes les places sont des files FIFO ce qui permet d’uniformiser le modèle. Nous verrons que nos réseaux, appelés réseaux FIFO colorés stricts (strict Colored FIFO Nets), abrégé en sCFN, sont à la fois une sous classe des réseaux FIFO (FIFO nets) et des réseaux de Petri colorés standard (Colored Petri Nets).

6.2.1 Réseaux FIFO Colorés stricts

Les réseaux FIFO ont été introduits par [90]. Dans ce type de réseaux de Petri les places sont des files d’attente. Une version hybride pour le réseau de Petri coloré a été introduit par [40, 122]. Dans cette version particulière, certaines places peuvent être des files d’attente de type FIFO d’autres non. Pour notre approche par composants ComSA, l’ordre des messages doit être préservé. Par conséquent, toutes les places de nos réseaux sont des files FIFO et nous imposons qu’un seul jeton soit consommé ou produit sur chaque arc. Il s’agit d’une restriction des réseaux FIFO colorés que nous appelons réseaux FIFO colorés stricts.

Un réseau FIFO coloré strict (sCFN) est un graphe dirigé biparti connecté. Il contient des places représentées par un cercle qui sont des files d’attente de type FIFO et des transitions représentées par un rectangle. Les places et les transitions sont reliées par des arcs. Les places décrivent les états du système et les transitions décrivent les actions. Chaque place est une file FIFO de marqueurs appelés jetons. Contrairement aux réseaux de Petri ordinaires où les jetons sont banalisés, ils constituent pour les réseaux de Petri colorés des entités distinctes. Cela permet, entre autres de :

- suivre leurs déplacements dans le réseau ;
- leur attribuer des propriétés qui peuvent varier en cours de simulation ;
- valider la même transition à partir de plusieurs jetons différents.

Un réseau FIFO coloré strict évolue lorsqu'on exécute une transition. Un jeton est alors retiré dans chacune des places en entrée de la transition et selon l'ordre de la file FIFO de la place. Des jetons sont déposés en fin des files d'attente des places en sortie de la transition. L'exécution d'une transition pour un réseau FIFO coloré strict est une opération indivisible.

Dans les sCFN, nous allons nous restreindre à des transitions qui consomment et produisent un seul jeton par arc. Un jeton est couvert par une valeur de données, qui appartient à un type donné. Un arc joignant une place et une transition est étiqueté par une expression qui est une fonction de couleur déterminant comment un jeton est consommé par une place ou produit dans une place lors du franchissement d'une transition. Ainsi les expressions arc décrivent comment l'état du sCFN change lorsque les transitions se déclenchent. La figure 6.1 représente les différents éléments composant un réseau sCFN.

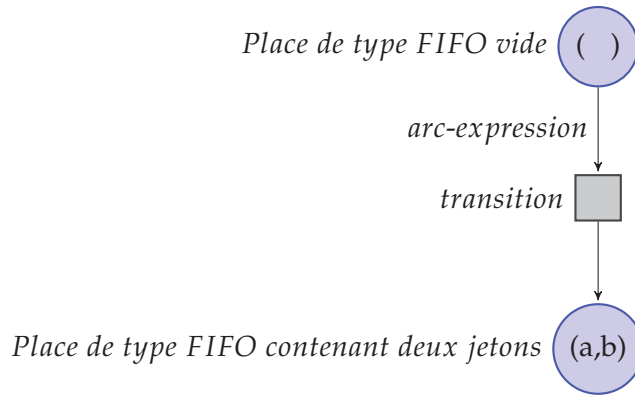


FIGURE 6.1 – Les différents éléments des réseaux sCFN.

Définition 17 Formellement, un réseau FIFO coloré strict est un tuple $sCFN = (P, T, A, I)$ où :

- P un ensemble non vide et fini de places qui sont des files d'attente de type FIFO ;
- T un ensemble non vide et fini de transitions, avec $P \cap T = \emptyset$ et nous affectons pour chaque $t \in T$ un poids W ;
- A un ensemble non vide et fini d'arcs, avec une fonction E qui associe à chaque arc a une expression, appelée arc-expression, du type de la place de l'arc a ;
- I est une fonction d'initialisation qui associe à certaines places un jeton du type de la place.

6.2.1.1 Le fonctionnement des réseaux FIFO colorés stricts

Dans le but d'illustrer un réseau sCFN, une file FIFO est une liste d'éléments (a_1, \dots, a_n) où a_1 est le dernier élément et a_n est le premier élément de la même liste. La liste vide est notée \emptyset . Un arc sortant d'une place p vers une transition t est exprimé par $\langle p, t \rangle$ et si son arc-expression e est défini, cet arc peut être exprimé par

$\langle p, t, e \rangle$. Un arc sortant d'une transition t vers une place p est exprimé par $\langle t, p \rangle$ et si son arc-expression e est défini, cet arc peut être exprimé par $\langle t, p, e \rangle$.

Pour chaque place p nous pouvons affecter des jetons représentant des données à l'aide de la fonction d'initialisation I en utilisant le symbole \mapsto , par exemple, pour la place p qui contient les jetons a et b , nous écrivons $p \mapsto (a, b)$.

Pour une transition t , nous appelons *places d'entrée* de t l'ensemble des places de telle sorte qu'il existe un arc $\langle p, t, e \rangle$ de A . Cet ensemble est défini comme suit :

$$\bullet t = \{p \in P \text{ tel que : } \langle p, t, e \rangle \in A\}$$

Symétriquement, nous appelons *places de sortie* de t l'ensemble des places de telle sorte qu'il existe un arc $\langle t, p, e \rangle$ de A . Cet ensemble est défini comme suit :

$$t \bullet = \{p \in P \text{ tel que : } \langle t, p, e \rangle \in A\}.$$

Systématiquement, l'ensemble des *transitions d'entrée* d'une place p est défini comme suit :

$$\bullet p = \{t \in T \text{ tel que : } \langle t, p, e \rangle \in A\}.$$

Et l'ensemble des *transitions de sortie* d'une place p est défini par :

$$p \bullet = \{t \in T \text{ tel que : } \langle p, t, e \rangle \in A\}.$$

La figure 6.2(a) représente le sCFN $\langle P, T, A, I \rangle$ où :

- $P = \{P_1, P_2, P_3\}$;
- $T = \{t_1, t_2\}$ avec $W(t_1) = 2$ et $W(t_2) = 1$;
- $A = \{\langle P_1, t_1, x_1 \rangle, \langle P_3, t_1, x_2 \rangle, \langle t_1, P_2, \max(x_1, x_1) \rangle, \langle P_2, t_2, x_1 \rangle, \langle P_3, t_2, x_2 \rangle, \langle t_2, P_3, x_1 + x_2 \rangle\}$;
- $I = \{P_1 \mapsto (1), P_2 \mapsto (5), P_3 \mapsto (2, 4)\}$.

Dans cette figure, par exemple, les jetons sont des nombres de telle sorte que les arc-expressions $\max(x_1, x_2)$ et $x_1 + x_2$ sont valides.

Dans cet exemple les places d'entrée de t_2 sont P_2 et P_3 , la place de sortie de t_2 est P_3 .

Un sCFN évolue lors de l'exécution d'une transition. Cela peut être réalisé que si les places d'entrée de toutes les transitions ne sont pas vides. Ensuite, le plus ancien jeton est retiré de chaque place d'entrée, et un jeton est placé dans les places de sortie de chaque transition.

Chaque jeton doit correspondre à l'expression qui marque l'arc reliant la place vers la transition. Un jeton est ensuite ajouté à chaque place de sortie de la transition.

La valeur de ce jeton est le résultat de l'évaluation de l'arc-expression de la transition et la place de sortie. Si deux transitions avec une place commune peuvent être appliquées, celle avec le plus grand poids est déclenchée. Si elles ont le même poids, une seule sera déclenchée aléatoirement.

Dans le sCFN de la figure 6.2(a), les deux transitions ont des places d'entrée et $W(t_1) > W(t_2)$, donc, la transition t_1 peut être appliquée. Cette transition prend le

jeton 1 de la place P_1 et le jeton 2 de la place P_3 . En effet, nous avons $x_1 \mapsto 1$ et $x_2 \mapsto 2$ la transition envoie $\max(x_1, x_2)$, c-à-d 2, vers la place P_2 et l'état suivant de ce réseau est illustré à l'aide de la figure 6.2(b). Par contre, maintenant la transition t_1 ne peut pas s'exécuter car la place P_1 est vide. Le franchissement de la transition t_2 prend le jeton 4 de la place P_3 et le jeton 5 de la place P_2 et produit la somme au niveau de la place P_3 . Le réseau sCFN résultat est représenté figure 6.2(c). La dernière étape génère le réseau sCFN illustré figure 6.2(d).

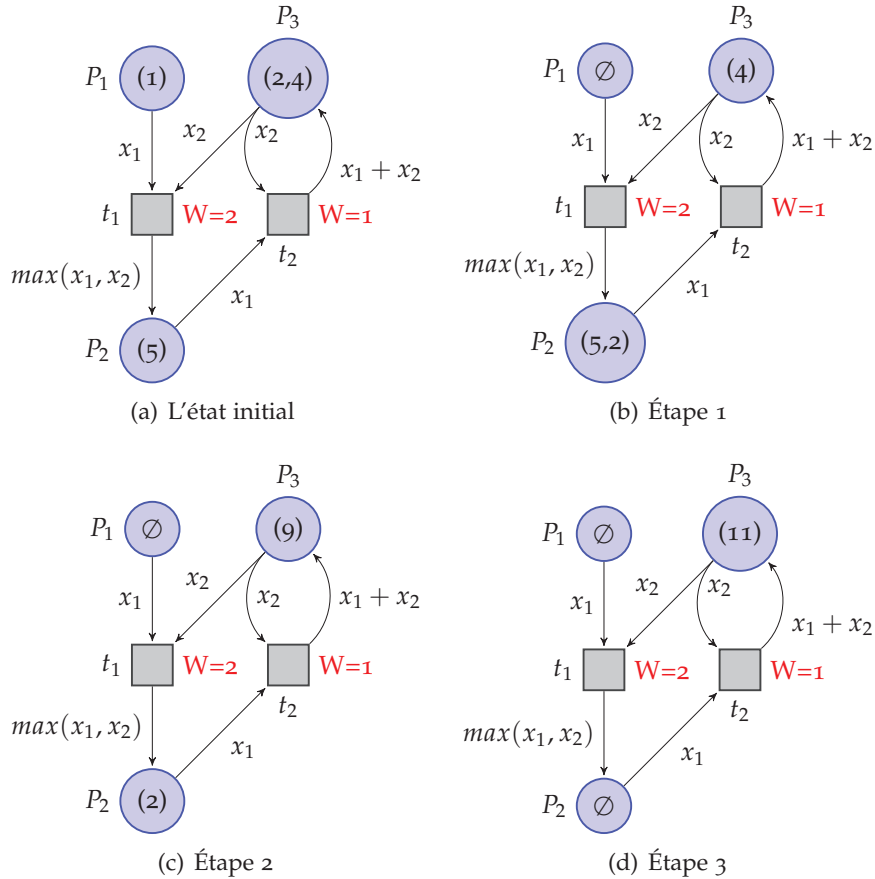


FIGURE 6.2 – Exemple d'un réseau sCFN et son évolution.

6.2.1.2 La sémantique des réseaux FIFO colorés stricts

Pour formaliser l'évolution d'un sCFN, une substitution σ de domaine V est définie où V est un ensemble fini de variables typées. Cette substitution est une application qui associe à chaque élément x de V une expression du type de x .

L'état d'un sCFN cfn est défini par une fonction qui indique le contenu de chacune des places du sCFN. Une telle fonction est appelée un *marquage*.

Définition 18 *Chaque place d'un réseau sCFN peut contenir un ou plusieurs jetons. La configuration complète du réseau forme un marquage en se basant sur tous les jetons positionnés dans ces places. Un marquage d'un réseau sCFN peut être un ensemble vide \emptyset .*

Une transition $t \in T$ est *activée* pour un marquage M avec la substitution σ si pour tout arc de la forme $\langle p, t \rangle$ de A on a $M(p) \neq \emptyset$ et $M(p)\uparrow = \sigma(E(\langle p, t \rangle))$. En d'autres termes une transition est activée si toutes ses places d'entrée p ne sont pas vides et que le prochain élément de leur file d'attente est compatible avec l'arc-expression de l'arc $\langle p, t \rangle$. Le franchissement de la transition va consister à consommer le jeton le plus ancien de chacune des places d'entrée de la transition et à ajouter un nouveau jeton dans chacune des places de sortie de la transition. Plus formellement, pour un sCFN cfn , soit $t \in T$, M un marquage et σ une substitution tels que t est activée pour M avec σ , on note $Eat_{t,\sigma}^{cfn}(M)$ le marquage M' tel que :

$$M'(p) = \begin{cases} M(p) - \forall p \in P \text{ t.q. } \langle p, t \rangle \in A \\ M(p) \text{ sinon} \end{cases}$$

et l'application de la transition t est notée $Next_{t,\sigma}^{cfn}(M)$, c'est le marquage M'' tel que :

$$M''(p) = \begin{cases} Eat_{t,\sigma}^{cfn}(M) + \sigma(e) \forall p \in P \text{ t.q. } \langle t, p, e \rangle \in A \text{ et } E(\langle t, p \rangle) = e \\ Eat_{t,\sigma}^{cfn}(M) \text{ sinon} \end{cases}$$

Le marquage initial d'un sCFN est le marquage M calculé à partir de la fonction d'initialisation, tel que :

$$M(p) = \begin{cases} I(p) \text{ si } \exists p \mapsto e \in I \\ \emptyset \text{ sinon} \end{cases}$$

Une transition $t \in T$ du sCFN cfn est *activable* pour un marquage M s'il existe une substitution σ telle que t est activée pour M avec σ . L'évolution d'un sCFN consiste à appliquer successivement des transitions activables à partir d'un marquage M . Lorsque plusieurs transitions ayant des places d'entrée communes sont activables pour un marquage M , la transition avec le plus grand poids est appliquée.

Les sCFN que nous venons de définir sont en fait une sous classe des réseaux de Petri colorés (Colored Petri Nets, CPN) comme l'illustre la figure 6.3. La transformation d'un sCFN en réseau de Petri coloré standard consiste simplement à considérer des places simples (qui ne sont pas des files FIFO) et à remplacer la couleur c de chaque jeton du sCFN par la couleur c de la liste L c-à-d d'explicitement indiquer que les places contiennent des listes de jetons. Une variable de type liste est ajoutée à l'ensemble des variables. Les expressions e des arcs allant d'une place vers une transition sont remplacées par l'expression $e :: L$ qui indique que pour effectuer une transition il faut prendre le premier élément de la liste stockée dans la place. Cette transformation peut se voir entre la place P_3 et la transition t_1 sur la figure 6.3. Les expressions e des arcs allant d'une transition à une place sont remplacées par l'expression $L\hat{~}[e]$ qui indique que le jeton produit par la transition doit être stocké en fin de liste de la place

destination. Cela est illustré sur la figure 6.3 par la transition t_1 et la place P_3 . Enfin, lorsqu'il y a un arc aller entre une place et une transition mais pas d'arc retour, cet arc est ajouté avec l'étiquette L afin de permettre à la transition de mettre à jour ou de consulter la liste de la place. Sur l'exemple de la figure 6.3(b), on voit l'ajout d'un arc entre t_1 et P_1 et entre P_2 et t_1 .

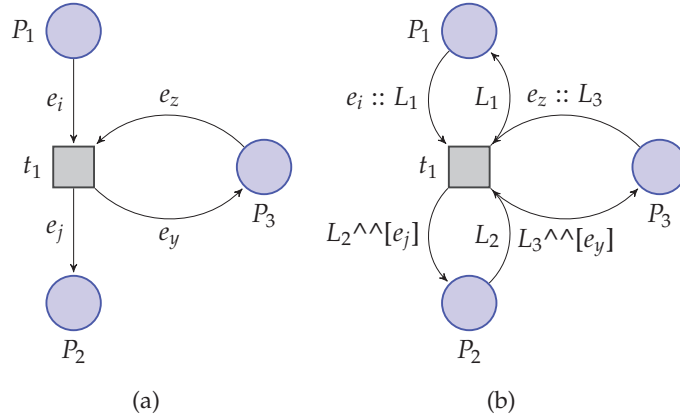


FIGURE 6.3 – (a) représente un sCFN et (b) son CPN équivalent.

La formalisation de notre approche par composants en sCFN consiste à définir séparément le sCFN des composants, des connecteurs puis enfin à combiner ces sCFN pour construire celui de l'application. Pour automatiser ces constructions nous avons besoin de trois structures supplémentaires qui apparaîtront soit lors de la construction de l'application soit dans la traduction d'un composant ou d'un connecteur en sCFN. Les sCFN de ces trois structures sont illustrés figure 6.4. Il s'agit d'exprimer les opérations suivantes :

- la *duplication* de données ou de signaux permettant de reproduire un signal ou un message vers plusieurs destinations. Comme illustré au niveau de la figure 6.4(a), la transition t_1 est activable si la place P_1 contient un jeton qui représente le message ou le signal à dupliquer. La transition t_1 consomme alors ce jeton et l'envoie dupliqué vers toutes ses places de sortie ;
- La *synchronisation* sert à produire un seul signal à partir de plusieurs signaux. Comme illustré par la figure 6.4(b) lorsque toutes les places P_1 , P_2 et P_3 sont marquées, la transition t_1 est activée. Elle consomme alors ces jetons et envoie un seul jeton de même type vers la seule place de sortie ;
- L'*incrément* du compteur de l'itération du composant est exprimée par le sCFN illustré par la figure 6.4(c). Ainsi si la place P_1 est marquée par un jeton de type entier, la transition t_1 est activable. Elle reçoit alors l'entier, l'incrémente et envoie la dernière valeur calculée sous forme d'un jeton vers la place P_1 .

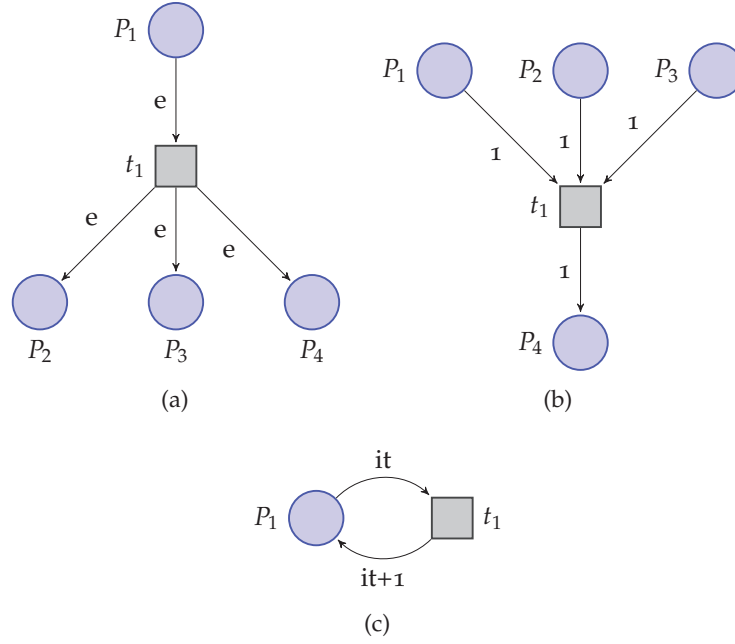


FIGURE 6.4 – Structures sCFN de : (a) la duplication, (b) la synchronisation et (c) l'itération.

6.2.2 Modèle des composants en sCFN

L'idée générale de la transformation est la suivante : les ports d'entrée et de sortie du composant seront représentés par des places, et chaque comportement du composant sera modélisé par une transition. Comme déjà vu dans la section 5.2, un composant effectue une itération dès que tous les ports d'entrée d'une de ses relations d'incidence contiennent des messages et toutes les relations d'incidence applicables sont déclenchées lors d'une itération. Pour traduire ce comportement en sCFN, nous allons considérer toutes les combinaisons de déclenchements possibles du composant et créer une transition pour chacune de ces combinaisons. Ces transitions auront pour poids le nombre de places en entrée afin d'être sûr de déclencher toutes les relations d'incidence applicables. Pour représenter le numéro d'itération associé à chaque message, nous allons avoir besoin de définir un compteur permettant de simuler ces numéros d'itération. Ce compteur sera implémenté par une place particulière et utilisera la structure définie par la figure 6.4(c).

Définition 19 L'ensemble des comportements possibles d'un composant C se définit à partir de l'ensemble des parties non vides de IR_C , appelé $\mathcal{P}(IR_C)$, par $B_C^{in} = \{IR^{in}(E) | E \in \mathcal{P}(IR_C)\}$.

Remarque 1 On peut noter que deux parties différentes de IR_C peuvent avoir le même ensemble de ports d'entrée, par exemple si $IR_C = \{r_1, r_2\}$ avec $r_1 = \{\langle i_1, i_2 \rangle, \{o_2\}\}$ et $r_2 = \{\langle i_2 \rangle, \{o_1\}\}$, les ensembles $\{r_1\}$ et $\{r_1, r_2\}$ ont les mêmes ports d'entrée. Cela signi-

fié simplement que lorsque r_1 est déclenchable alors r_2 l'est aussi et donc nous allons produire une seule transition avec comme entrées i_1 et i_2 dans le réseau de Petri associé au composant. Cette transition aura en sortie o_1 et o_2 .

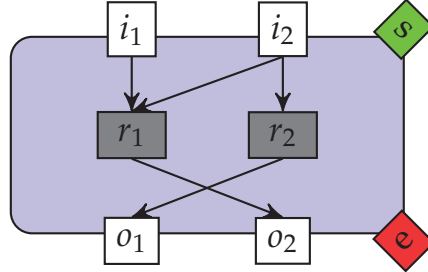


FIGURE 6.5 – Représentation du composant de la remarque 1.

Dans les algorithmes suivants la notation $AddArc(a, e, A, E)$ où a est un arc, e une expression d'arc, A un ensemble d'arcs et E un ensemble d'arc-expressions, sera un raccourci pour $A := A \cup \{a\}$; $E := E \cup \{a \mapsto e\}$;

La fonction de transformation d'un composant C est définie par l'algorithme 1. L'application de cet algorithme au composant de la remarque 1 est illustrée dans la figure 6.6.

Définition 20 Soient $App = (Comp \cup Conn, DI \cup TI)$ une application et $C \in Comp$ un composant de l'application. Soit $sCFN_C$ la fonction de transformation du composant C en $sCFN$, alors $sCFN_C(C) = \langle P_C, T_C, A_C, I_C \rangle$ où P_C est l'ensemble des places, T_C l'ensemble des transitions, A_C l'ensemble des arcs et I_C la fonction d'initialisation de l'ensemble P_C .

Dans l'algorithme 1, la place P_p est associée pour chaque port p du composant C . D'autres places P_{it} et P_{es} sont utilisées pour mémoriser respectivement le numéro d'itération du composant et gérer le passage d'une itération à l'autre (ligne 1). Par conséquent :

$$P_C = P_C^{IR} \cup P_C^{Sig}$$

où $P_C^{IR} = \{P_p | p \in pIn_C \cup pOut_C\}$ et $P_C^{Sig} = \{P_e, P_s\} \cup \{P_{it}, P_{es}\}$.

Un composant itère uniquement lorsque tous les ports d'au moins une de ses relations d'incidence d'entrée contiennent des messages. Ensuite, toutes les relations d'incidence satisfaites sont déclenchées. Pour traduire ce comportement en $sCFN$, une transition t est créé pour toutes les combinaisons possibles de relations d'incidence satisfaites avec son poids (ligne 6). Dans l'exemple 6.6, les deux transitions $t_{i_1 i_2}$ et t_{i_2} modélisent les deux comportements du composant de la figure 6.5.

Les deux transitions t_e et t_s sont ajoutées à la transformation du composant (ligne 15). La transition t_e permet la duplication du signal de la fin de l'itération vers le reste de l'application et la transition t_s permet la synchronisation avec les signaux en provenance du reste de l'application. Dans tous les cas, le signal de la fin d'itération

se redirige de P_e vers P_{es} et de P_{es} vers P_s pour se préparer à une nouvelle itération. Donc :

$$T_C = T_C^{IR} \cup T_C^{Sig}$$

où $T_C^{IR} = \{t_i | i \in B_C^{in}\}$ et $T_C^{Sig} = \{t_e, t_s\}$.

Le déclenchement d'une transition $t_i \in T_C^{IR}$ avec $i \in B_C^{in}$ consomme un message dans toutes les places d'entrée et produit un message dans toutes les places de sortie (lignes 9 et 11). Les arcs correspondants sont définis comme suit :

$$A_C^P = \{\langle P_p, t_i, \langle x_p, m_p \rangle \rangle | P_p \in P_C^{IR}, p \in i\} \text{ et } A_C^T = \bigcup_{E \in \mathcal{P}(\mathcal{IR}_C)} \{\langle t_i, P_p, \langle x_{it}, m_p \rangle \rangle | i \in IR^{in}(E), p \in IR^{out}(E)\}.$$

où l'expression sur les jetons est un couple avec une variable pour le numéro d'itération du composant et le message du composant.

Dans une application, les synchronisations des composants sont gérées à l'aide des ports s et e de chaque composant. Pour modéliser ces contrôles, la place P_s doit être marquée pour déclencher une transition $t_i \in T_C^{IR}$ avec $i \in B_C^{in}$ qui doit alors envoyer un signal notifiant la fin d'une itération vers la place P_e (lignes 7 et 14). Si la transition t_i est déclenchée, elle doit également mettre à jour le numéro d'itération. D'abord, elle envoie le numéro d'itération actuel vers la place P_{it} , ensuite elle reçoit le numéro d'itération incrémenté (ligne 12). De plus, pour représenter la boucle « *wait-get-put* », il faut envoyer le signal de la place P_e vers la place P_s (lignes 16, 17, 18 et 19). Par conséquent, les arcs supplémentaires sont définis par :

$$\begin{aligned} A_C^{P_e} &= \{\langle t_i, P_e, 1 \rangle | t_i \in T_C^{IR}\}, \\ A_C^{P_s} &= \{\langle P_s, t_i, 1 \rangle | t_i \in T_C^{IR}\}, \\ A_C^{it} &= \{\langle P_{it}, t_i, x_{it} \rangle, \langle t_i, P_{it}, x_{it} + 1 \rangle | t_i \in T_C^{IR}\} \text{ et } \\ A_C^{Sig} &= \{\langle P_e, t_e, 1 \rangle, \langle t_e, P_{es}, 1 \rangle, \langle P_{es}, t_s, 1 \rangle, \langle t_s, P_s, 1 \rangle\}. \end{aligned}$$

Donc, l'ensemble A_C est l'union des ensembles précédents :

$$A_C = A_C^P \cup A_C^T \cup A_C^{P_e} \cup A_C^{it} \cup A_C^{Sig}$$

Nous allons montrer que le réseau $sCFN_C(C)$ simule bien le comportement du composant C tel que défini dans la section 5.4.

Pour un marquage M , nous allons noter $P_{in}(\vec{M}(C))$ le vecteur de messages compatible avec pIn_C tel que :

$$P_{in}(\vec{M}(C))|_p = \begin{cases} M(p) \uparrow & \text{si } M(p) \neq \emptyset \\ \emptyset & \text{sinon} \end{cases}$$

Et de la même manière, on définit $P_{out}(\vec{M}(C))$ le vecteur de messages compatible avec $pOut_C$.

Soit M un marquage tel que $M(P_s) = 1$, $M(P_{it}) = n$, $M(P_e) = M(P_{es}) = \emptyset$. On ne fait pas d'hypothèse sur les places qui représentent les ports de sortie car étant

Algorithme 1 : La fonction $sCFN_C$ pour un composant**Entrées :** C un composant.**Sorties :** (P, T, A, I) $sCFN$ et E ensemble d'arc-expressions.

// Une place par port, une place pour le numéro d'itération et une place pour le changement d'itération

1 $P := \{P_p | p \in Port_C\} \cup \{P_{it}, P_e, P_s, P_{es}\};$ 2 $T := \emptyset;$ 3 $A := \emptyset;$ 4 $E := \emptyset;$

// Ajout d'une transition pour chaque comportement

5 **Pour chaque** $B \in B_C^{in}$ **Faire**6 $T := T \cup \{t_B\};$ // P_s doit contenir le signal pour activer la transition7 $AddArc(\langle P_s, t_B \rangle, 1, A, E);$ // Consommation d'un message sur chaque port d'entrée de B 8 **Pour chaque** $i \in B$ **Faire**9 $AddArc(\langle P_i, t_B \rangle, \langle x_i, m_i \rangle, A, E);$ // Production d'un message sur chaque port de sortie de $IR^{out}(B)$ 10 **Pour chaque** $o \in IR^{out}(B)$ **Faire**11 $AddArc(\langle t_B, P_o \rangle, \langle x_{it}, m_o \rangle, A, E);$

// Incrémentation du numéro d'itération

12 $AddArc(\langle P_{it}, t_B \rangle, x_{it}, A, E);$ 13 $AddArc(\langle t_B, P_{it} \rangle, x_{it} + 1, A, E);$

// Envoi du signal de fin d'itération

14 $AddArc(\langle t_B, P_e \rangle, 1, A, E);$

// Ajout des transitions de changement d'itération

15 $T := T \cup \{t_e, t_s\};$ 16 $AddArc(\langle P_e, t_e \rangle, 1, A, E);$ 17 $AddArc(\langle t_e, P_{es} \rangle, 1, A, E);$ 18 $AddArc(\langle P_{es}, t_s \rangle, 1, A, E);$ 19 $AddArc(\langle t_s, P_s \rangle, 1, A, E);$

// On initialise toutes les places des ports de données par la liste vide, la place gérant le numéro d'itération à 0 et enfin la place d'activation à 1.

20 $I := \{P \mapsto [] | P \notin \{P_{it}, P_e, P_s\}\} \cup \{P_{it} \mapsto 0, P_s \mapsto 1\};$

// Le poids de chaque transition est le nombre de place d'entrée de la transition

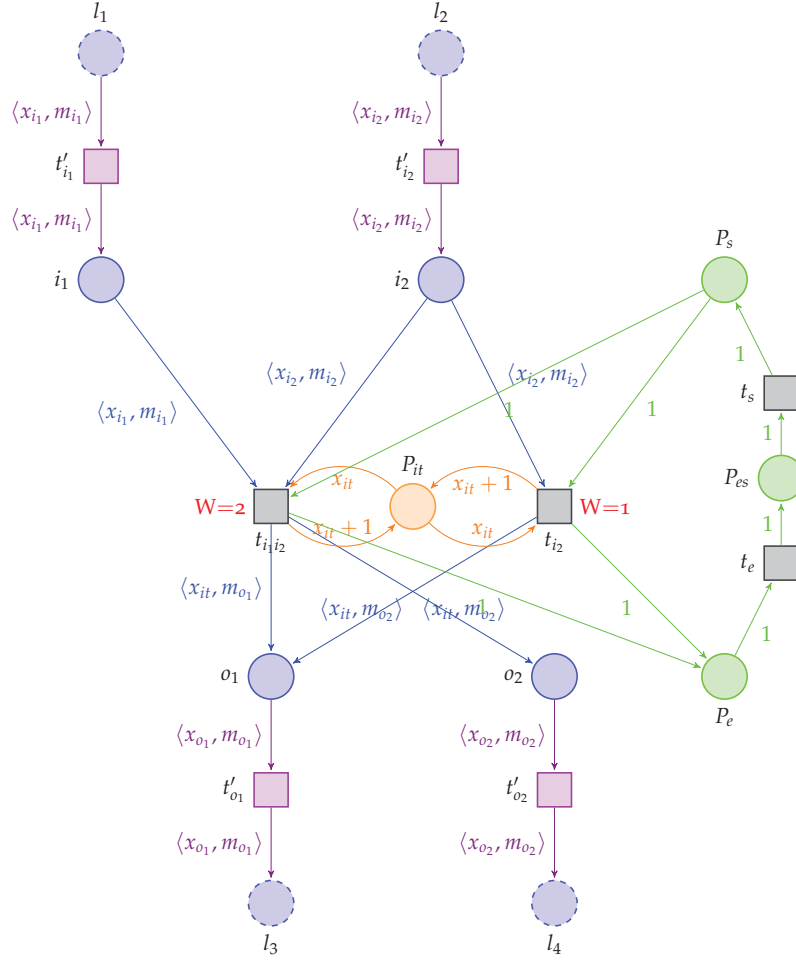


FIGURE 6.6 – $sCFN_C(C)$ pour le composant de la remarque 1. Les places l_1 , l_2 , l_3 , l_4 ainsi que les transitions t'_{i_1} , t'_{i_2} , t'_{o_1} et t'_{o_2} ne font pas partie de $sCFN_C(C)$ mais représentent comment les ports du composant sont reliés à l'extérieur.

des files FIFO, les messages qui y seront stockés seront tous restitués dans leur ordre d'arrivée dans le réseau, nous n'avons donc pas besoin de nous assurer que ces places soient vides pour commencer une itération. Par contre on peut constater que, par construction, aucune transition de $sCFN_C(C)$ ne sera activable si $P_{in}(\vec{M}(C))$ ne valide aucune relation d'incidence de C . Par contre, si $P_{in}(\vec{M}(C))$ valide au moins une relation d'incidence alors, d'après la sémantique des $sCFN$, la transition activable ayant le plus fort poids sera appliquée c-à-d celle qui simule toutes les relations validées par $P_{in}(\vec{M}(C))$.

L'application de cette transition va enlever un message dans chacune des places qui représentent des ports d'entrée des relations validées et placera un nouveau message estampillé du numéro d'itération courant dans toutes les places qui représentent les

ports de sortie des relations validées. L'application de la transition va aussi ajouter 1 à la place P_{it} et placer le signal 1 dans le port P_e . Après l'application de cette transition, aucune transition représentant les comportements ne pourra être appliquée car la place P_s est vide. La seule transition activable est t_e qui va placer le signal dans la place P_{es} . Cette transition permet aussi de dupliquer ce message si le port e est connecté dans l'application où est utilisé le composant. Puis la transition t_s va replacer le signal dans la place P_s . Cette transition permettra de synchroniser la place P_s avec l'arrivée de signaux venant d'autres éléments de l'application si le port s est connecté. Ce cycle est bien conforme au comportement du composant défini dans la section 5.4.

6.2.3 Modèle des connecteurs en sCFN

La transformation du connecteur se réalise, comme pour un composant, en représentant les ports par des places et les comportements internes du connecteur par des transitions. Les connecteurs ont un port de données entrée, un port de données sortie et un port de déclenchement s nécessaires pour communiquer avec les autres éléments de l'application, sauf le connecteur *sFIFO* qui a juste deux ports de données.

La modélisation en sCFN d'un connecteur c est définie par la fonction $sCFN_c(c)$

Définition 21 Soient $App = (Comp \cup Conn, DI \cup TI)$ une application et $c \in Conn$ un connecteur de l'application. Soit $sCFN_c$ qui désigne la fonction de transformation de c en sCFN, alors $sCFN_c(c) = \langle P_c, T_c, A_c, I_c \rangle$ où P_c est l'ensemble des places, T_c est l'ensemble des transitions, A_c l'ensemble des arcs et I_c la fonction d'initialisation de l'ensemble P_c .

La modélisation en sCFN de chaque connecteur est représentée dans les figures 6.7, 6.8 et 6.9.

La fonction de transformation $sCFN_c(c) = \langle P_c, T_c, A_c, I_c \rangle$ de nos connecteurs en sCFN peut facilement être déduite de cette représentation graphique.

Le connecteur *sFIFO* permet d'acheminer une donnée sans la stocker. Si un composant C_y est connecté en aval avec un connecteur *sFIFO* alors il envoie un signal à partir de son port e vers le port s du composant C_x connecté en amont du connecteur *sFIFO* lorsque son itération est terminée. Alors lorsque C_x fournit une donnée au port d'entrée du connecteur *sFIFO*, cette donnée est modélisée par un jeton placé sur F_{in} du réseau FIFO coloré strict du *sFIFO*, sachant que par définition de *sFIFO* la place F_{in} est vide. La présence du jeton dans cette place va rendre la transition t_F activable. Cette transition va consommer ce jeton et le déposer dans la place F_{out} . Le comportement, fixé au niveau de la définition du *sFIFO*, est représenté par une seule transition, figure 6.7, et elle correspond juste à l'acheminement des données.

Pour les connecteurs de type *Buffer* le $sCFN_c$ doit modéliser la bufferisation des messages. Ainsi ils sont modélisés à partir de deux transitions pour respectivement consommer les données qui arrivent et extraire le message à envoyer en sortie. Ainsi la disponibilité d'une donnée sur le port d'entrée d'un connecteur est modélisé par l'existence d'un jeton dans la place B_{in} . Ce jeton va être transporté par la transition t_{B_1}

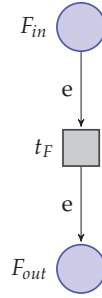
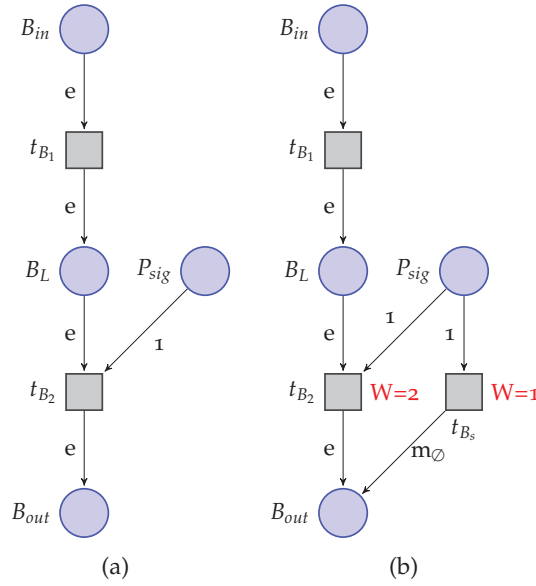


FIGURE 6.7 – sCFN du connecteur sFIFO.

vers la place B_L qui va stocker le nouveau jeton et le restituer plus tard dans le bon ordre.

La délivrance d'une donnée au niveau du port de sortie B_{out} se réalise s'il existe un jeton dans la place P_{sig} . Cette place correspond au port s des connecteurs de type *Buffer*. Les jetons qui existent dans la place P_{sig} sont de type entier. Cette place représente le port s des connecteurs et la présence d'un jeton dans P_{sig} ainsi que la présence d'un jeton dans la place B_L déclenchent la transmission d'un jeton vers la place B_{out} .

FIGURE 6.8 – Réseau FIFO coloré strict du connecteur *bBuffer* (a) et le connecteur *nbBuffer* (b).

Pour le connecteur en mode *bloquant* c-à-d le connecteur *bBuffer*, l'absence de jetons dans la place B_L ou dans la place P_{sig} rend la transition t_{B2} non activable comme illustré par la figure 6.8(a).

Pour le connecteur en mode *non bloquant* c-à-d le connecteur *nbBuffer*, la place P_{sig} est également liée à la place B_{out} pour permettre au connecteur de générer des messages vides m_{\emptyset} grâce à la transition t_{Bs} , comme illustré par la figure 6.8(b). Mais si les

deux places B_L et P_{sig} sont marquées, les deux transitions t_{B_2} et t_{B_s} sont alors activables. La sémantique de $sCFN_c$ va déterminer que seule la transition t_{B_2} est déclenchée car son poids est 2 ce qui la rend prioritaire par rapport à t_{B_s} .

Le $sCFN_c$ des connecteurs de type *Greedy*, quant à lui, doit modéliser la perte des messages.

Ainsi la disponibilité d'une donnée sur le port d'entrée d'un connecteur est modélisé par l'existence d'un jeton dans la place G_{in} . Ce jeton va être transporté par la transition t_{G_1} vers la place G_L qui contient un jeton de type liste, ne stocke que le dernier jeton reçu en écrasant l'ancien.

La place P_{sig} fonctionne de la même façon que dans les connecteurs de type *Buffer* et elle correspond au port s des connecteurs de type *Greedy*. Si cette place contient un jeton et si la place G_L contient un jeton différent de la liste vide, alors une donnée est délivrée au niveau de la place de sortie G_{out} .

Pour le connecteur en mode *bloquant bGreedy*, l'absence de jetons c-à-d la liste est vide dans la place G_L ou dans la place P_{sig} , rend la transition t_{G_2} non activable comme illustré par la figure 6.9(a).

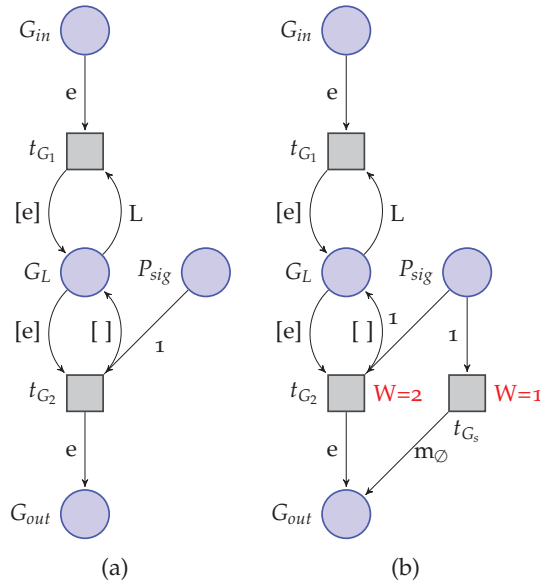


FIGURE 6.9 – Réseau FIFO coloré strict du connecteur *bGreedy* (a) et le connecteur *nbGreedy* (b).

Pour le connecteur en mode *non bloquant nbGreedy*, la place P_{sig} est également liée à la place G_{out} pour permettre au connecteur de générer des messages vides m_\emptyset grâce à la transition t_{G_s} , comme illustré par la figure 6.9(b). Mais si les deux places G_L et P_{sig} sont marquées, les deux transitions t_{G_2} et t_{G_s} sont alors activables. La sémantique de $sCFN_c$ va déterminer que seule la transition t_{G_2} est déclenchée car son poids est 2 ce qui la rend prioritaire par rapport à t_{G_s} .

6.2.4 Modèle d'une application en sCFN

Il s'agit maintenant de décrire comment se modélise un graphe d'application en sCFN. Pour cela nous allons définir la fonction $sCFN_G$, représentée par l'algorithme 2, pour transformer un graphe d'application G défini par $(Comp \cup Conn, DI \cup TI)$. Il s'agit principalement de modéliser les liens de $DI \cup TI$. Cette opération s'effectue en trois étapes :

1. la transformation des composants et des connecteurs en sCFN ;
2. l'ajout aux composants des transitions permettant la duplication des messages en sortie et l'ajout de messages en entrée pour la synchronisation ;
3. la connexion proprement dite des liens de $DI \cup TI$ sur ces transitions.

Avant toute chose, nous allons considérer que tous les composants et tous les connecteurs ont des noms de port différents, pour cela il suffit de considérer par exemple que chaque composant et chaque connecteur a un identifiant unique et que le nom de chacun des ports de ces objets est préfixé par cet identifiant.

Définition 22 Soit $App = (Comp \cup Conn, DI \cup TI)$ une application. Soit $sCFN_G$ la fonction de transformation de App en sCFN, alors $sCFN_G(App) = \langle P, T, A, I \rangle$ où P est l'ensemble des places, T est l'ensemble des transitions, A est l'ensemble des arcs et I la fonction appliquée pour associer une liste FIFO à chaque place.

La première étape de la transformation consiste à appliquer respectivement $sCFN_C(C)$ et $sCFN_c(c)$ sur les composants de $Comp$ et les connecteurs de $Conn$. Ensuite, l'union de ces sCFN se fait selon la définition 23. En effet, l'algorithme 2 définit la transformation d'un graphe d'application en sCFN. Cet algorithme permet d'automatiser la modélisation des applications ComSA.

Définition 23 Soit $sCFN_1 = \langle P_1, T_1, A_1, I_1 \rangle$ et $sCFN_2 = \langle P_2, T_2, A_2, I_2 \rangle$ deux sCFN avec P_1, T_1, P_2 et T_2 des ensembles finis disjoints de places et de transitions. L'union de $sCFN_1$ et $sCFN_2$ notée $sCFN_1 \cup sCFN_2$ est le tuple $\langle P_1 \cup P_2, T_1 \cup T_2, A_1 \cup A_2, I_1 \cup I_2 \rangle$.

Donc, la première transformation (ligne 1) définit les ensembles de places, de transitions et d'arcs comme de l'union des sCFN des composants et des connecteurs :

$$\langle P_U, T_U, A_U, I_U \rangle = \left(\bigcup_{C \in Comp} sCFN_C(C) \right) \cup \left(\bigcup_{c \in Conn} sCFN_c(c) \right).$$

La deuxième étape (ligne 9) définit l'ensemble de transitions T_I pour réaliser les connexions entre les composants et les connecteurs et l'ensemble d'arcs A_I pour modéliser leurs communications (ligne 10). Ainsi, pour chaque port d'entrée connecté p d'un composant C , une transition t est ajoutée afin de placer un message dans la place P_p correspondante.

$$T_I = \bigcup_{C \in Comp} \{t_p | p \in pIn_C \text{ s.t. } \exists \langle x, C^p \rangle \in DI\}.$$

$$A_I = \bigcup_{C \in \text{Comp}} \{ \langle t_p, P_p, \langle x_p, m_p \rangle \rangle \mid p \in pIn_C \text{ s.t. } \exists \langle x, C^p \rangle \in DI \}.$$

Réciproquement, la deuxième étape (lignes 12 et 13) définit également les ensembles T_O et A_O pour créer la transition t_p (et un arc) pour chaque port de sortie p d'un composant C . Cette transition t_p permet de recevoir et de diffuser, si nécessaire, un message produit et placé dans la place P_p .

$$T_O = \bigcup_{C \in \text{Comp}} \{ t_p \mid p \in pOut_C \}.$$

$$A_O = \bigcup_{C \in \text{Comp}} \{ \langle P_p, t_p, \langle x_p, m_p \rangle \rangle \mid p \in pOut_C \}.$$

Ces ensembles sont illustrés par la figure 6.6 respectivement par les transitions t'_{i_1} et t'_{i_2} et les transitions t'_{o_1} et t'_{o_2} .

Afin de déclencher un connecteur à partir de plusieurs signaux de fin d'itération, une transition t_s est ajoutée pour chaque place P_s du connecteur. Il mène aux deux ensembles T_s (ligne 4) et A_s (ligne 5) suivants :

$$T_s = \bigcup_{c \in \text{Conn}} \{ t_s \}, A_s = \bigcup_{c \in \text{Conn}} \{ \langle t_s, P_s, 1 \rangle \}.$$

Le signal de la fin d'itération d'un composant peut être utilisé par plusieurs synchronisations. En effet, la transition t_e est utilisée pour dupliquer le signal reçu à partir de la place P_e du sCFN d'un composant. Mais, il faut ajouter une place (ligne 15) pour chaque lien $l \in TI$ impliquant un port e pour compléter ce signal de duplication. Le signal du port e est envoyé vers le port s d'un composant ou d'un connecteur (ligne 16). Notons $P_{\langle e,s \rangle}$ la place supplémentaire liée au lien $\langle C^e, x^s \rangle$ où x est un composant ou un connecteur. Cela ramène aux deux ensembles T_{sig} et A_{sig} où a désigne soit un composant soit un connecteur :

$$P_{sig} = \bigcup_{l = \langle C^e, a^s \rangle \in TI} \{ P_{\langle e,s \rangle} \}, A_{sig} = \bigcup_{l = \langle C^e, a^s \rangle \in TI} \{ \langle t_e, P_{\langle e,s \rangle}, 1 \rangle \}.$$

La dernière étape de la transformation consiste à connecter les liens de $DI \cup TI$ avec les transitions. Cela est représenté par deux ensembles A_{DI} et A_{TI} . Les arcs définis par l'ensemble A_{DI} (lignes 19 et 21) complètent les communications des données entre les composants et les connecteurs. L'ensemble A_{TI} (ligne 17) définit les arcs dédiés aux communications des signaux :

$$A_{DI} = \left(\bigcup_{l = \langle C^p, c^i \rangle \in DI} \{ \langle t_p, P_i, \langle x_p, m_p \rangle \rangle \} \right) \cup \left(\bigcup_{l = \langle c^o, C^p \rangle \in DI} \{ \langle P_o, t_p, \langle x_p, m_p \rangle \rangle \} \right).$$

$$A_{TI} = \left(\bigcup_{l = \langle C_1^e, C_2^s \rangle \in TI} \{ \langle P_{\langle e,s \rangle}, t_s, 1 \rangle \} \right) \cup \left(\bigcup_{l = \langle C^e, c^s \rangle \in TI} \{ \langle P_{\langle e,s \rangle}, t_s, 1 \rangle \} \right).$$

Algorithme 2 : La fonction $sCFN_G$ d'un graphe d'application

Entrées : Un graphe d'application ($Comp \cup Conn, Dl \cup Tl$)
Sorties : $sCFN (P, T, A, I)$

```

// Étape 1
1 Soit  $sCFN_{\cup}(Comp \cup Conn) = (P, T, A, I)$ ;
// Étape 2
2 Pour chaque  $c \in Conn$  Faire
3   si  $Type(c) \neq sFIFO$  alors
4      $T := T \cup \{t_s, t_o\}$ ;
5      $AddArc(\langle t_s, p_s \rangle, 1, A, E)$ ;
6      $AddArc(\langle p_o, t_o \rangle, 1, A, E)$ ;
    // où  $s$  est le port de déclenchement de  $c$ 
7 Pour chaque  $C \in Comp$  Faire
    // Pour chaque port d'entrée connecté de  $C$  on ajoute la transition
    // permettant d'ajouter un message dans la place
8   Pour chaque  $p \in \overline{pIn}_C$  Faire
9      $T := T \cup \{t_p\}$ ;
10     $AddArc(\langle P_p, t_p \rangle, \langle x_p, m_p \rangle, A, E)$ ;
    // Pour chaque port de sortie de  $C$  on ajoute la transition
    // permettant de diffuser le message produit dans le réseau
11   Pour chaque  $p \in pOut_C$  Faire
12      $T := T \cup \{t_p\}$ ;
13     $AddArc(\langle t_p, P_p \rangle, \langle x_p, m_p \rangle, A, E)$ ;
// Étape 3
14 Pour chaque  $\langle e, s \rangle \in Tl$  Faire
    // Création d'une place pour accueillir le signal émis par  $e$ 
15    $P := P \cup \{P_{es}\}$ ;
    // Duplication du signal émis par  $e$  vers l'extérieur
16    $AddArc(\langle t_e, P_{\langle e, s \rangle} \rangle, 1, A, E)$ ;
    // Transmission du signal vers la transition de synchronisation du
    // destinataire
17    $AddArc(\langle P_{\langle e, s \rangle}, t_s \rangle, 1, A, E)$ ;
    // Transmission des messages vers les ports d'entrée des composants
18 Pour chaque  $\langle s, d \rangle \in InComp(Dl)$  Faire
19    $AddArc(\langle p_s, t_d \rangle, \langle x_d, m_d \rangle, A, E)$ ;
    // Transmission des messages vers les ports d'entrée des connecteurs
20 Pour chaque  $\langle s, d \rangle \in InConn(Dl)$  Faire
21    $AddArc(\langle t_s, P_d \rangle, \langle x_s, m_s \rangle, A, E)$ ;

```


Finalement, $sCFN_G(App) = \langle P, T, A, I \rangle$ est défini par $P = P_U \cup P_{sig}$, $T = T_U \cup T_I \cup T_O \cup T_s$, $A = A_U \cup A_I \cup A_O \cup A_s \cup A_{sig} \cup A_{DI} \cup A_{TI}$ et $I = I_U$.

Pour comprendre facilement l'algorithme 2, nous allons définir des ensembles qui vont être utiles afin de transformer un graphe d'application en sCFN et distinguer les différents types de liens.

Pour un ensemble de composants et de connecteurs \mathcal{C} , le réseau $sCFN_U(\mathcal{C})$ dénote le sCFN défini par $\bigcup_{C \in \mathcal{C}} sCFN(C)$. La première étape de l'algorithme consiste donc simplement à calculer $sCFN_U(Comp \cup Conn)$.

Considérons un composant C de $Comp$, on note :

- \overline{pIn}_C l'ensemble des ports de données d'entrée connectés de C dans G , c-à-d $\overline{pIn}_C = \{p_{in} \in pIn_C \mid \exists p_{out}, \langle p_{out}, p_{in} \rangle \in DI\}$;
- $\overline{pIn}(G)$ l'ensemble de tous les ports d'entrée connectés de l'application et se définit par $\overline{pIn}(G) = \bigcup_{C \in Comp} \overline{pIn}_C$.

Les liens de données vont être partitionnés pour séparer les liens aboutissant au port i d'un connecteur et les liens aboutissant à un port d'entrée d'un composant de la manière suivante :

- $InConn(DI) = \{\langle p_1, p_2 \rangle \in DI \mid \exists c \in Conn, p_2 \in pIn(c)\}$;
- $InComp(DI) = \{\langle p_1, p_2 \rangle \in DI \mid \exists C \in Comp, p_2 \in pIn(C)\}$.

De même, nous allons distinguer les liens déclencheurs aboutissant sur le port s d'un composant et les liens déclencheurs aboutissant sur le port s d'un connecteur. Nous noterons ces deux ensembles $InConn(TI)$ et $InComp(TI)$.

Pour montrer que $sCFN_G(App)$ modélise bien la sémantique définie dans la section 5.4, il suffit d'observer le comportement des réseaux représentant les liens entre les éléments de l'application. Lorsqu'un message est disponible sur un port p de données en sortie d'un composant ou d'un connecteur, alors la transition t_p correspondante est activable, donc le message pourra franchir la transition et aller se placer dans toutes les places de sortie de t_p . Lorsqu'un composant émet un signal sur son port e , ce signal sera dupliqué dans toutes les places P_{es} par la transition t_s du composant, puis ce signal sera acheminé dans la place P_s du composant ou dans la place P_{sig} du connecteur grâce à la transition t_s correspondante. Cette dernière transition ne sera activable que si un signal se trouve dans chacune de ses places d'entrée ce qui effectuera la synchronisation. Par conséquent, étant donné que $sCFN_C(C)$ modélise la sémantique de C pour tout $C \in Comp$ et $sCFN_c(c)$ modélise la sémantique de c pour tout $c \in Conn$. De plus, tous les sCFN modélisant les liens $sCFN_G(App)$ sont conformes à la sémantique des liens dans App , alors, le réseau FIFO coloré strict $sCFN_G(App)$ modélise bien le comportement de l'application App .

6.3 L'ANALYSE DES BLOCAGES DES APPLICATIONS COMSA

L'analyse des blocages de nos applications est basée sur la propriété de vivacité des places de sCFN. Cette propriété exprime qu'à partir de chaque marquage accessible M et chaque place p , il existe un marquage M' qui marque p et accessible depuis M .

Cette section présente une condition suffisante pour assurer la vivacité des places des sCFN. Ensuite, elle montre comment ce résultat peut être utilisé pour aider les utilisateurs à construire une application sans blocages.

6.3.1 La vivacité des sCFN

Le concept de vivacité des sCFN est étudié par l'absence totale de blocages dans les applications représentées par les réseaux FIFO colorés stricts. Une application bien formée n'implique pas qu'elle est *vivante*. L'absence de blocages est basée sur la vivacité des places des sCFN. Cette propriété exprime que tous les ports des composants dans un graphe d'application reçoivent des données durant l'exécution de l'application.

L'idée générale de l'étude de la vivacité des sCFN consiste à simuler les sCFN par les réseaux de Petri ordinaires (Ordinary Petri Nets, OPN) et par les réseaux à choix asymétrique (Asymmetric Choice Nets, ACN). La propriété de la vivacité des places des sCFN utilise une condition suffisante de la vivacité des places des réseaux OPN.

Définition 24 Un réseau de Petri ordinaire OPN est un tuple $\langle P, T, A, M \rangle$ où P est un ensemble fini de places, T est un ensemble fini de transitions avec $P \cap T = \emptyset$, A un ensemble d'arcs sous la forme $\langle p, t \rangle$ ou sous la forme $\langle t, p \rangle$ où $p \in P$, $t \in T$. Finalement, M est une fonction d'assignement d'un nombre positif de jetons pour chaque place. La notation $\langle P, T, A \rangle$ se réfère à la structure du réseau $\langle P, T, A, M \rangle$.

Les notations suivantes sont utilisées au niveau de l'évolution des réseaux de Petri ordinaire OPN. Dans un OPN $\langle P, T, A, M \rangle$:

- L'ensemble des places E est *marqué* si $\exists p \in E$ tel que $M(p) > 0$;
- Une place p est appelée *place source* si $\bullet p = \emptyset$.

Dans un OPN $\langle P, T, A, M \rangle$, une transition $t \in T$ est *active* si $\forall p \in \bullet t \ M(p) > 0$. Franchir une transition active consiste à consommer un jeton de chaque place d'entrée et produire un jeton dans chaque place de sortie. L'OPN $\langle P, T, A, M \rangle$ peut évoluer vers $\langle P, T, A, M' \rangle$ lors du franchissement de la transition t noté :

$$\langle P, T, A, M \rangle \rightarrow_t \langle P, T, A, M' \rangle$$

si t est une transition active de $\langle P, T, A, M \rangle$ et M' est le marquage défini comme suit :

$$M_R(p) = \begin{cases} M(p) - 1 & \text{si } p \in \bullet t \\ M(p) & \text{sinon} \end{cases} \quad M'(p) = \begin{cases} M_R(p) + 1 & \text{si } p \in t \bullet \\ M_R(p) & \text{sinon} \end{cases}$$

M_R représente un marquage auxiliaire qui aide à définir M' notamment pour les places $t \bullet \cap \bullet t$. Pour un OPN $\langle P, T, A, M \rangle$, le marquage M' est dit *accessible en une seule étape* à partir du marquage M si $\langle P, T, A, M \rangle \rightarrow_t \langle P, T, A, M' \rangle$ pour tout $t \in T$, il est noté :

$$M \rightarrow_{\langle P, T, A \rangle} M' \text{ ou } M \rightarrow M'$$

La fermeture réflexive et transitive de \rightarrow est notée \rightarrow^* . Le marquage M' est *accessible* à partir du marquage M si $M \rightarrow^* M'$.

Un OPN $\langle P, T, A, M \rangle$ est *vivant* si pour tout marquage M' accessible depuis M et pour toute transition $t \in T$, il existe un marquage M'' accessible à partir de M' tel que t est activée dans $\langle P, T, A, M'' \rangle$. De plus, il est *place-vivant* si pour chaque marquage M' accessible depuis M et pour chaque place $p \in P$, il existe un marquage M'' qui marque p .

Dans la suite, tous les réseaux OPN contiennent au moins une place et une transition qui sont connectés afin de ne pas tenir compte de plusieurs sous-réseaux non connectés dans les démonstrations c-à-d qu'ils ne contiennent pas de place isolée.

Nous présentons quelques définitions et les propriétés liées à la structure d'un OPN.

Définition 25 Soit $\langle P, T, A, M \rangle$ un OPN. Un ensemble de places $E \subseteq P$ est appelé *siphon* si ses prédécesseurs sont aussi des successeurs, c-à-d :

$$\bullet E \subseteq E \bullet$$

Définition 26 Soit $\langle P, T, A, M \rangle$ un OPN. Un ensemble de places $E \subseteq P$ est appelé *trappe* si ses successeurs sont aussi des prédécesseurs, c-à-d :

$$E \bullet \subseteq \bullet E$$

Définition 27 Un OPN $\langle P, T, A, M \rangle$ est un ACN si :

$$\forall p_1, p_2 \in P \quad p_1 \bullet \cap p_2 \bullet \neq \emptyset \implies p_1 \bullet \subseteq p_2 \bullet \text{ ou } p_2 \bullet \subseteq p_1 \bullet.$$

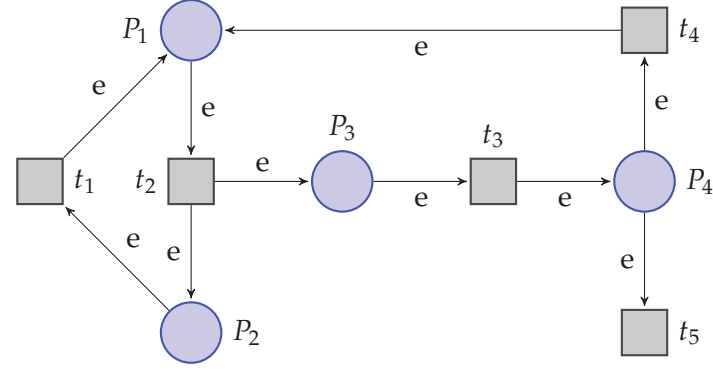
Ces définitions sont très importantes pour prouver la vivacité d'un ACN. Si un siphon ne contient pas de jeton pour un marquage M , alors il reste non marqué pour chaque marquage M' accessible à partir de M . Symétriquement, si une trappe est marquée pour un marquage M alors elle reste marquée pour chaque marquage M' accessible à partir de M .

Si tout siphon contient une trappe initialement marquée, on dit que le réseau de Petri vérifie *le théorème de Commoner*. Cette propriété est une condition suffisante de vivacité pour certaines classes de réseau de Petri à savoir les réseaux ACN. La figure 6.10 illustre un exemple d'un siphon E contenant une trappe Q .

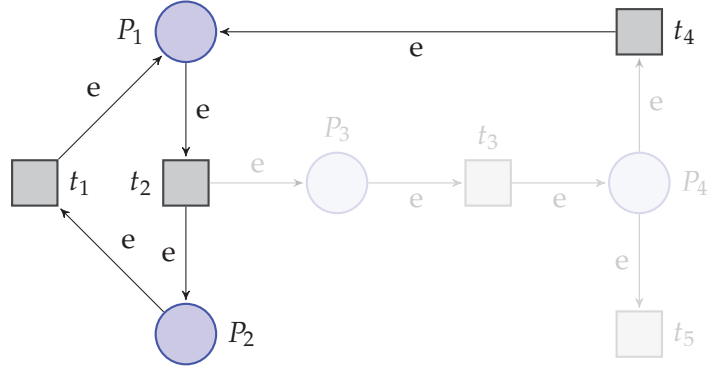
Théorème 1 (théorème de Commoner) Si tous les siphons d'un ACN contiennent une trappe marquée, alors ce réseau est vivant.

En utilisant le théorème 1, nous allons garantir une condition suffisante pour la vivacité des réseaux ACN. Pour cela, nous allons réaliser une simulation des sCFN par les OPN de telle manière que les places des sCFN contenant des jetons sont les mêmes places des OPN.

D'abord, un OPN peut avoir deux marquages M et M' qui peuvent être équivalents. À ce stade, l'OPN en question a le même ensemble des transitions activées. Dans



(a) Exemple d'un siphon $E = \{p_1, p_2, p_3, p_4\}$, $\bullet E = \{t_1, t_2, t_3, t_4\}$, $E\bullet = \{t_1, t_2, t_3, t_4, t_5\}$.



(b) Exemple d'une trappe $Q = \{p_1, p_2\}$, $Q\bullet = \{t_1, t_2\}$, $\bullet Q = \{t_1, t_2, t_4\}$.

FIGURE 6.10 – Exemple d'un siphon E contenant une trappe Q .

ce cas, nous considérons deux OPN N et N' , avec N' un sous-réseau de N , M marquage de N et M' marquage de N' . Il s'agit, ensuite, de prouver que si l'OPN N' est vivant alors l'OPN N est place-vivant.

Définition 28 Soit $N = \langle P, T, A \rangle$ un OPN, M et M' sont deux marquages de N tels que $M(p) > 0 \Leftrightarrow M'(p) > 0$, alors M et M' sont appelés marquage-équivalent. Cette équivalence est notée $M \leftrightarrow M'$ et si $M \leftrightarrow M'$ alors $\langle P, T, A, M \rangle$ et $\langle P, T, A, M' \rangle$ ont les mêmes transitions activées.

Définition 29 Soit $N = \langle P, T, A \rangle$ et $N' = \langle P', T', A' \rangle$ deux OPN tels que N, N' ne contiennent pas de place isolée. N' est un sous-réseau de N si $P' \subseteq P$, $T' \subseteq T$ et $A' \subseteq A$ avec $\forall \langle x, y \rangle \in A'$ $x, y \in P' \cup T'$. N' est appelé sous-jacent de N s'il est un sous-réseau de N tel que $P' = P$ et $A' = \{\langle x, y \rangle \in A \mid x \in T' \text{ ou } y \in T'\}$.

Lemme 1 Soit $N = \langle P, T, A \rangle$ un OPN et $N' = \langle P, T', A' \rangle$ un OPN sous-jacent de N .

Soit M_1 un marquage tel que $\langle P, T', A', M_1 \rangle$ est vivant, si $\forall M_2$ tel que $\langle P, T, A, M_1 \rangle \rightarrow_t \langle P, T, A, M_2 \rangle$ nous avons $\langle P, T', A', M_2 \rangle$ vivant alors $\langle P, T, A, M_1 \rangle$ est place-vivant.

Démonstration. Prouver que $\langle P, T, A, M_1 \rangle$ est place-vivant ramène à prouver que $\forall M_2$ tel que $M_1 \rightarrow_N^* M_2$, $\forall p \in P$, $\exists M_3$ tel que $M_2 \rightarrow_N^* M_3$ et $M_3(p) > 0$.

Premièrement, nous remarquons que si $\langle P, T, A, M_1 \rangle$ où M_1 est tel que $\langle P, T', A', M_1 \rangle$ est vivant alors $\forall p \in P \exists M_2$ tel que $M_2(p) > 0$ et $M_1 \rightarrow_N M_2$.

En effet, étant donné que $\langle P, T', A', M_1 \rangle$ est vivant, alors, il est place-vivant, i.e. $\forall p \in P \exists M_2$ tel que $M_1 \rightarrow_{N'} M_2$ et $M_2(p) > 0$. Puisque N' est sous-jacent de N toutes les transitions utilisées dans $M_1 \rightarrow_{N'} M_2$ sont des transitions de N , donc, $M_1 \rightarrow_N M_2$.

Par hypothèse si $\langle P, T', A', M_1 \rangle$ est vivant alors $\forall M_2$ tel que $\langle P, T, A, M_1 \rangle \rightarrow_t \langle P, T, A, M_2 \rangle$, M_2 vérifie $\langle P, T', A', M_2 \rangle$ est vivant. Ainsi, par induction, il est évident que tout marquage M_3 accessible à partir du marquage M_1 rend N' vivant. En effet, $\langle P, T', A', M_3 \rangle$ est vivant. Ainsi, $\forall p \in P \exists M_4$ accessible depuis M_3 qui marque p , donc $\langle P, T, A, M_1 \rangle$ est place-vivant. \square

Nous allons définir une classe de réseaux OPN qui préserve la vivacité des réseaux sous-jacent ACN. Pour cela certaines définitions supplémentaires doivent être introduites.

Définition 30 Soit $\langle P, T, A \rangle$ un OPN. La transition $t \in T$ est nommée couvrante si $\exists T' \subset T$ tel que $t \notin T'$, $\bullet t = \bullet T'$ et $t \bullet = T' \bullet$. Dans ce cas, nous disons que t couvre T'

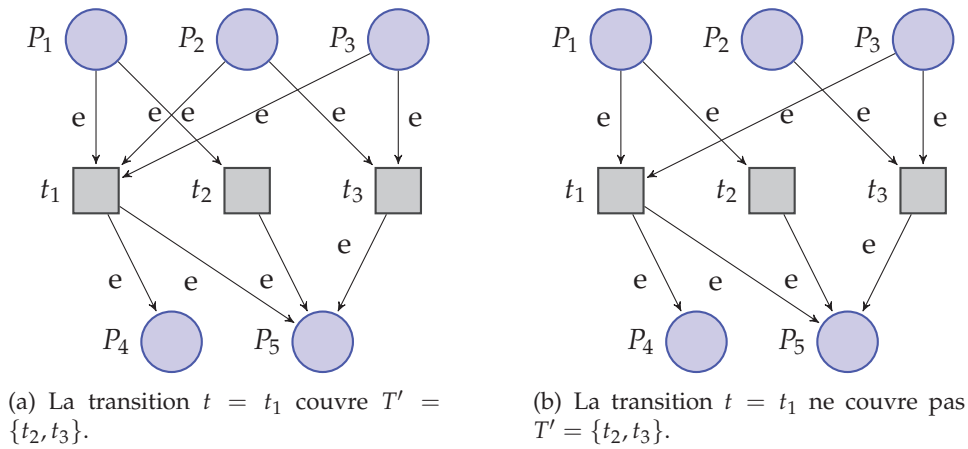


FIGURE 6.11 – L'illustration d'une transition couvrante et d'une transition non couvrante.

La figure 6.11 montre une illustration d'une transition couvrante et d'une transition non couvrante. Prenons $t = t_1$ et $T' = \{t_2, t_3\}$ avec $t \notin T'$, la figure 6.11(a) montre que $\bullet t = \bullet T'$ car $\bullet t = \{P_1, P_2, P_3\}$ et $\bullet T' = \{P_1, P_2, P_3\}$. De plus $t \bullet = T' \bullet$ car $t \bullet = \{P_4, P_5\}$ et $T' \bullet = \{P_4, P_5\}$. En effet, la transition $t = t_1$ couvre $T' = \{t_2, t_3\}$. Par contre, la figure 6.11(b) montre que $t = t_1$ ne couvre pas $T' = \{t_2, t_3\}$ car $\bullet t \neq \bullet T'$.

Lemme 2 Soit $\langle P, T, A, M \rangle$ un OPN où t couvre $T' \subset T$ et M' tel que $\langle P, T, A, M \rangle \rightarrow_t \langle P, T, A, M' \rangle$ alors $\exists \bar{M} \leftrightarrow M$ et $\bar{M}' \leftrightarrow M'$ tel que $\langle P, T, A, \bar{M} \rangle \rightarrow^* \langle P, T, A, \bar{M}' \rangle$ où chaque transition de T' est franchie une fois dans la dérivation.

Démonstration. Soit $\langle P, T, A, M \rangle \rightarrow_t \langle P, T, A, M' \rangle$, M' fait en sorte que $M'(p) > 0 \forall p \in t\bullet$ et $M'(p) = M(p) - 1 \forall p \in \bullet t \setminus t\bullet$.

$$\text{Soit } M_t \text{ un marquage tel que } M_t(p) = \begin{cases} M(p) - 1 + |p\bullet \cap T'| & \text{si } p \in \bullet t \\ M(p) & \text{sinon} \end{cases}$$

Le franchissement de t depuis M ($\forall p \in \bullet t, M(p) > 0$) et M_t ajoute seulement des jetons dans $\bullet t$, alors $M \leftrightarrow M_t$. De plus, pour chaque place $p \in \bullet t$ nous avons $M_t(p)$ est plus grand que le nombre de transitions de T' dont p est une place d'entrée. En conséquence, il est possible de franchir une fois chaque transition de T' à partir de M_t . Le marquage résultat M'_t fait en sorte que $M'_t(p) > 0$ pour toute place $p \in T'\bullet$ avec t couvre T' , en effet, $T'\bullet = t\bullet$. Et $M'_t(p) = M_t(p) - |p\bullet \cap T'| = M(p) - 1$ pour toute place $p \in \bullet T' \setminus T'\bullet$. Ainsi, $M'_t(p) \leftrightarrow M'$. \square

Pour simuler la communication avec perte des connecteurs *bGreedy* et *nbGreedy* présentés dans la section 5.2. Un réseau spécifique est nécessaire nommé *réseau Lossy* et est défini par la définition suivante :

Définition 31 Un *réseau Lossy* est un OPN permettant de perdre des jetons tel que $\text{Lossy} = \langle P_L, T_L, A_L \rangle$ comme l'indique la figure 6.12(a). La transition t_l est nommée transition de perte.

La figure 6.12 montre que la place p_b est un buffer qui accepte juste un seul jeton. Ce mécanisme est contrôlé par la place p_o indiquant que le buffer est vide et par la place p_i indiquant que le buffer est plein. La transition t_l est utilisée pour ignorer les jetons arrivant dans p_e quand la place p_b contient un jeton. Plus précisément, commençons avec un marquage où la place p_o contient un jeton et les autres places sont vides, quand un jeton arrive dans la place p_e la transition t_e est activée, comme illustré par la figure 6.12(a). Une fois la transition t_e est franchie, un jeton est placé dans la place p_i et dans la place p_b par contre la place p_e est vide, comme illustré par la figure 6.12(b). En conséquence, la transition t_e n'est pas activée, mais dès qu'un jeton arrive à la place p_e , la transition t_l est activée, comme illustré par la figure 6.12(c). Tant que la transition t_o n'est pas franchie, les jetons arrivant dans la place p_e peuvent être ignorés en utilisant la transition t_l , comme illustré par la figure 6.12(d). Quand la transition t_o est activée, les places p_i et p_b deviennent vides et la place p_o contient un jeton qui indique que le prochain jeton qui arrivera dans la place p_e sera mis dans le buffer.

Un *réseau Lossy* OPN n'est pas un ACN puisque :

1. $p_e\bullet$ et $p_i\bullet$ contiennent t_l ;
2. $p_e\bullet \not\subseteq p_i\bullet$ et $p_i\bullet \not\subseteq p_e\bullet$.

Cependant, un *réseau Lossy* OPN préserve la vivacité du réseau ACN couvrant.

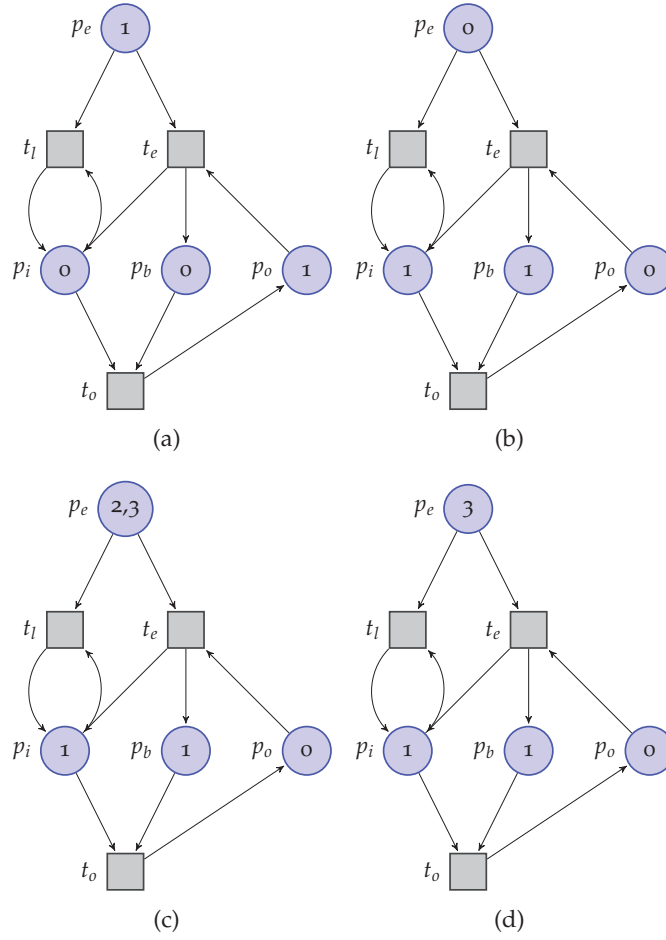


FIGURE 6.12 – (a) Le réseau Lossy (état initial), (b) étape 2 du réseau Lossy de la figure (a), (c) étape 3 du réseau Lossy de la figure (a) et (d) étape 4 du réseau Lossy de la figure (a).

Définition 32 Soit $N = \langle P, T, A \rangle$ un OPN, on dit que N contient le réseau Lossy $N' = \langle P_L, T_L, A_L \rangle$ qui est connecté par $\{p_e, t_o\}$, si les seuls arcs entre N' et $N \setminus N'$ sont les suivants $\{\langle t, p_e \rangle | t \in T \setminus T_L\}$, $\{\langle t_o, p \rangle | p \in P \setminus P_L\}$ et $\{\langle p, t_o \rangle | p \in P \setminus P_L\}$.

Lemme 3 Soit N un OPN contenant un sous-réseau N' qui est un réseau Lossy connecté par $\{p_e, t_o\}$. Soit M un marquage tel que $M(p_o) = 1$ et $M(p_i) = M(p_b) = 0$. Pour chaque marquage M' accessible depuis M alors $M'(p_b) = 1$ si la transition t_l est franchie.

Démonstration. Puisque $p_i \in \bullet t_l$, la transition t_l est activée signifie que la place p_i n'est pas vide. De plus, puisque $\bullet p_i = \{t_e\}$, p_i est marquée quand la transition t_e est franchie, aussi la place p_b est marquée puisque $p_b \in t_e \bullet$. La transition t_e ne peut pas être franchie si la place p_o est vide. Ainsi, les deux places p_i et p_b ne peuvent pas contenir plus d'un jeton. Quand la transition t_o est franchie, p_i et p_b sont vidées et

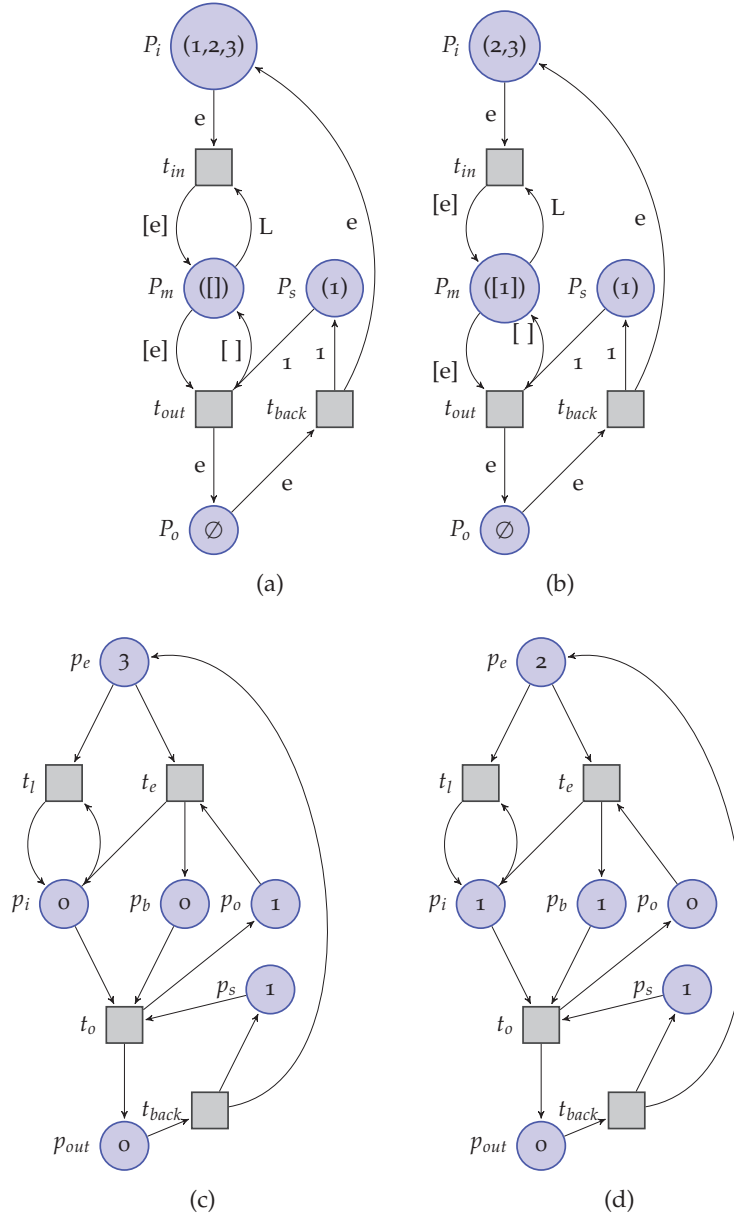


FIGURE 6.13 – (a) Exemple d'un sCFN (état initial), (b) étape 2 du sCFN de la figure (a), (c) Simulation du sCFN de la figure (a) et (d) Simulation du sCFN de la figure (b).

la place p_o reçoit un jeton. Ce qui implique que si $M(p_i) = M(p_b) = 0$ nous avons $M(p_o) = 1$ et si $M(p_i) = M(p_b) = 1$ nous avons $M(p_o) = 0$. \square

Définition 33 Un réseau étendu à choix asymétrique (extended asymmetric choice net, EACN) est un OPN $N = \langle P, T, A \rangle$ tel que le réseau sous-jacent $\langle P, T', A' \rangle$ est un ACN où

$T' = T \setminus (\{t \in T \mid t \text{ est couvrante}\} \cup \bigcup_{L \in \mathcal{L}(N)} \{t_L\})$ avec $\mathcal{L}(N)$ l'ensemble de tous les réseaux Lossy connectés par un ensemble $\{p_e, t_o\}$ de N .

Théorème 2 Soit $N = \langle P, T, A \rangle$ un EACN dont l'ACN sous-jacent est $N' = \langle P, T', A' \rangle$ et M_1 un marquage tel que $\langle P, T', A', M_1 \rangle$ marque une trappe dans chaque siphon de N' . Tout marquage M_2 tel que $\langle P, T, A, M_1 \rangle \rightarrow_t \langle P, T, A, M_2 \rangle$, marque une trappe dans chaque siphon de N' .

Démonstration. D'abord, notons que si N' est un ACN et M marque une trappe dans chaque siphon de N et en se basant sur la propriété de la vivacité d'un ACN, alors tout marquage accessible depuis M marque également une trappe appartenant à un siphon de N' . Ensuite, nous allons prouver le théorème 2. Trois cas doivent être distingués selon le type de la transition t :

- Si $t \in T'$, la propriété est vérifiée si N' est un ACN ;
- Si t est couvrante, alors en se basant sur le lemme 2 nous savons que $\langle P, T, A, M_1 \rangle \rightarrow_t \langle P, T, A, M_2 \rangle$ correspond à la dérivation $\langle P, T, A, \overline{M_1} \rangle \rightarrow_{N'}^* \langle P, T, A, \overline{M_2} \rangle$ tel que $M_1 \leftrightarrow \overline{M_1}$ et $M_2 \leftrightarrow \overline{M_2}$. Si $\overline{M_2}$ marque chaque trappe appartenant à un siphon de N' , alors M_2 vérifie aussi cette propriété ;
- Si t est une transition de perte, la seule place qui peut être vide après le franchissement de t est la place p_e associée à la transition t . Nous pouvons remarquer que chaque trappe de N' qui contient cette place p_e contient aussi la place p_i et la place p_b associées à t . Si la transition t est activée dans M_1 , d'après le lemme 3 les deux places p_i et p_b sont marquées dans M et elles sont aussi marquées dans M_2 . Ainsi, chaque trappe de N' marquée dans M_1 reste marquée dans M_2 . \square

Définition 34 Soit $N = \langle P, T, A \rangle$ un sCFN, $\overline{N} = \langle \overline{P}, \overline{T}, \overline{A} \rangle$ un OPN et les trois fonctions $\Phi_P : \overline{P} \rightarrow P$, $\Phi_T : \overline{T} \rightarrow T$ et $\Phi_M : \text{FIFO} \rightarrow \mathbb{N}$ où les deux dernières sont des applications. N' simule N par Φ_P, Φ_T et Φ_M si pour chaque marquage M_1 et M_2 $\langle P, T, A, M_1 \rangle \rightarrow_t \langle P, T, A, M_2 \rangle$ si et seulement si $\langle \overline{P}, \overline{T}, \overline{A}, \Phi_M(M_1) \rangle \rightarrow_{\Phi_T(t)} \langle \overline{P}, \overline{T}, \overline{A}, \Phi_M(M_2) \rangle$.

L'OPN de la figure 6.13(c) simule le sCFN de la figure 6.13(a) avec $\Phi_P = \{p_e \mapsto p_i, p_b \mapsto p_m, p_s \mapsto p_s, p_{out} \mapsto p_o\}$, $\Phi_T = \{t_l \mapsto t_{in}, t_e \mapsto t_{in}, t_o \mapsto t_{out}, t_{back} \mapsto t_{back}\}$ et $\Phi_M(x)$ est le nombre d'éléments de x pour chaque FIFO x sauf $([])$ où $\Phi_M([]) = 0$. La figure 6.13(d) montre le résultat du franchissement de la transition t_l qui correspond au réseau sCFN de la figure 6.13(b).

Théorème 3 Soit S un sCFN et N un OPN qui simule S par Φ_P, Φ_T et Φ_M , si N est place-vivant alors S est aussi place-vivant.

Démonstration. La démonstration du théorème 3 est évidente à partir de la définition 34 et du théorème 2. \square

Il n'est pas toujours possible de trouver un OPN simulant un sCFN. Par contre, les sCFN décrivant la sémantique de notre modèle par composants ComSA, peuvent être

simulés par les OPN en utilisant la généralisation des fonctions Φ_T , Φ_P et Φ_M décrites ci-dessus. Le réseau OPN obtenu appartient à la classe EACN quand les relations d'incidence des composants ComSA respectent la condition de choix asymétrique c-à-d $IR^{in}(r_1) \cap IR^{in}(r_2) \neq \emptyset \Rightarrow IR^{out}(r_1) \subseteq IR^{out}(r_2) \vee IR^{out}(r_2) \subseteq IR^{out}(r_1)$. En effet, il existe deux types de places ne vérifiant pas la condition ACN, à savoir les places modélisant les ports d'entrée des connecteurs et les places p_e des sous-réseaux Lossy. Dans les deux cas, les transitions appartenant à l'ensemble des transitions de sortie de la place sont couvrantes ou respectent la condition ACN.

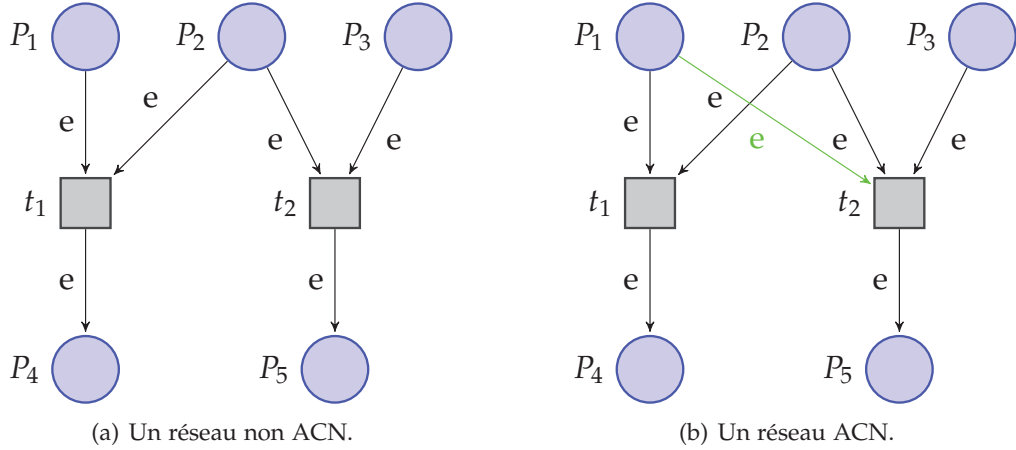


FIGURE 6.14 – Un exemple d'un réseau ACN et d'un réseau non ACN.

La figure 6.14 montre un exemple d'un réseau ACN et un autre exemple d'un réseau non ACN. Prenons par exemple les deux ensembles $E_1 = \{P_1, P_2\}$ et $E_2 = \{P_2, P_3\}$, comme illustré par la figure 6.14(a). Ces deux ensembles ne vérifient pas la condition de choix asymétrique, présentée par la définition 27, car $E_1 \bullet \not\subseteq E_2 \bullet$. Par contre, comme illustré par la figure 6.14(b), les deux ensemble E_1 et E_2 vérifient bien cette condition.

Si un sCFN est place-vivant, il peut ne pas être vivant. Cependant, les seules transitions qui ne sont pas vivantes sont celles représentant les compositions des comportements élémentaires des composants. Ceci ne pose aucun problème étant donné que chaque comportement élémentaire est vivant. D'autre part, le poids défini sur les transitions peut empêcher le franchissement de certaines transitions. Cela ne représente pas un vrai problème puisque dans le sCFN construit d'une application, si la transition t est en conflit avec une autre transition t' et $W(t) > W(t')$ alors t est couvrante. Cela signifie que le comportement modélisé par t contient le comportement modélisé par t' .

Cette analyse d'un EACN simulant un sCFN d'une application ComSA détecte si l'application peut être lancée et qu'elle est vivante c-à-d ne contient pas de blocages. Elle permet aussi d'indiquer les erreurs en vue d'être corrigées.

6.3.2 La configuration de démarrage des applications ComSA

Nous nous sommes particulièrement intéressés à garantir que les applications ComSA ne contiennent pas de blocages. Cela peut être fait en étudiant la place-vivacité des sCFN. Les résultats que nous obtenons nous permettent de définir une condition de démarrage qui assure la vivacité d'une application ComSA.

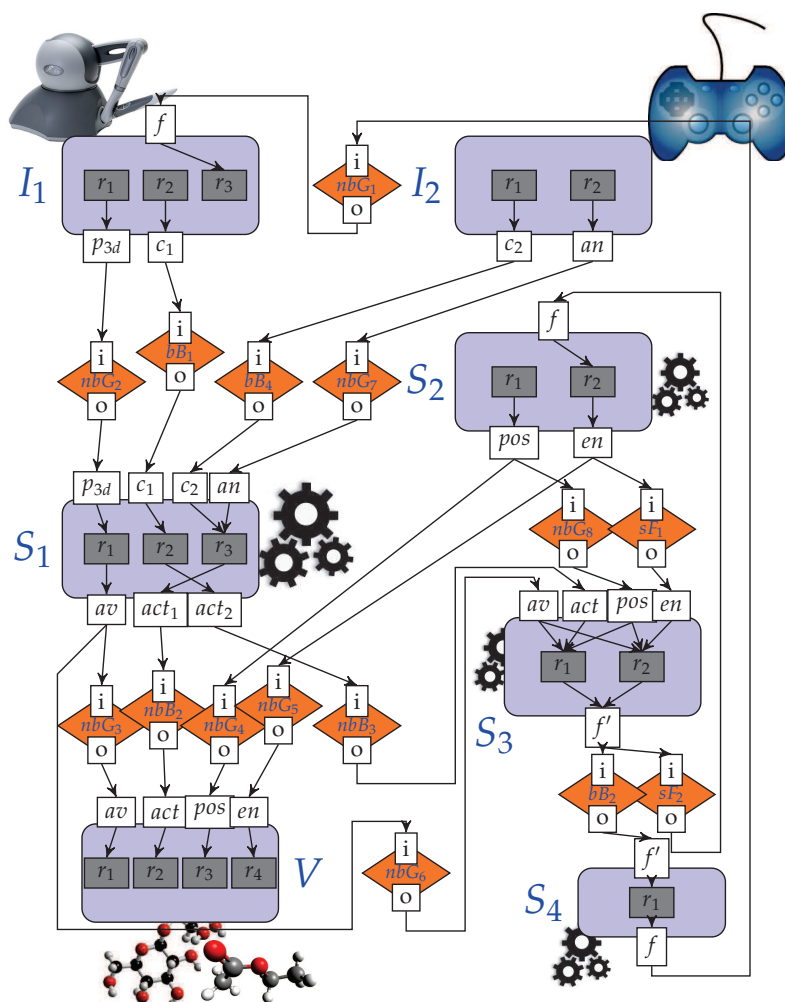


FIGURE 6.15 – Graphe d'une application de dynamique moléculaire.

Le processus de démarrage d'une application ComSA se compose de trois étapes principales et va être appliqué sur l'application représentée par la figure 6.15 :

1. **La condition ACN** : la première étape consiste à vérifier que l'ensemble des relations d'incidence actives d'une application respectent la condition ACN. Si non, nous proposons à l'utilisateur de verrouiller les comportements de composants concernés. Cela est équivalent, par exemple, à remplacer les deux relations

$\langle I_1, O_1 \rangle$ et $\langle I_2, O_2 \rangle$ par la relation $\langle I_1 \cup I_2, O_1 \cup O_2 \rangle$. En effet, les relations d'incidence non asymétriques sont éliminées. La conséquence de cette élimination est que l'application n'exploite pas toutes les synchronisations autorisées par le composant, mais nous pouvons éviter les blocages.

2. **La condition siphon-trappe** : la deuxième étape permet de vérifier que tous les siphons d'un sCFN d'une application contiennent une trappe. Si un siphon ne contient pas de trappe, le cycle formant ce siphon est identifié explicitement pour que l'utilisateur puisse modifier la structure de l'application pour la rendre vivante. Cette étape est NP-difficile et pour détecter les siphons contenant une trappe, nous avons utilisé la méthode décrit dans [158]. Cette méthode permet de traduire le problème original vers un problème SAT en se basant sur trois contraintes :

- (a) L'ensemble P des places est un siphon :

$$\bigvee_{p \in P} p^{(0)} \wedge \bigwedge_{t \in T} \bigwedge_{p \in t \bullet} (p^{(0)} \implies \bigvee_{p' \in \bullet t} p'^{(0)})$$

- (b) Max-trappe \subseteq siphon :

$$\bigwedge_{i=0}^n \bigwedge_{p \in P} (p^{i+1} \iff (p^{(i)} \wedge \bigwedge_{t \in T} \bigwedge_{p \in \bullet t} \bigvee_{p' \in t \bullet} p'^{(i)}))$$

- (c) Max-trappe est non marquée :

$$\bigvee_{p \in P: m_0(p) > 0} \neg p^{(n+1)}$$

La première contrainte permet de chercher tous les ensembles de places composant un siphon. La première partie de cette contrainte indique la non vacuité, par contre la deuxième partie représente la condition d'un siphon $\bullet P \subseteq P \bullet$ avec P un ensemble de places du sCFN.

La deuxième contrainte consiste à identifier tous les siphons contenant une trappe maximale. Pour un siphon P , nous pouvons calculer sa trappe maximale en supprimant toutes les places dont les transitions de sortie n'ont pas de places de sortie en P . Cette procédure ajoute $(n + 1)$ places $p^{(0)}, \dots, p^{(n)}$ pour chaque place p avec n le nombre de places dans un sCFN. Les variables $p^{(0)}$ représentent les siphons non vides comme mentionné par la première contrainte. Les variables $p^{(i)}$ représentent les étapes intermédiaires P_i de la procédure de la génération de la trappe maximale. P_{i+1} est obtenu à partir de P_i en supprimant toutes les places dont les transitions de sortie n'ont pas de places de sortie en P_i . Puisqu'il y a n places, la procédure converge après n itérations, ainsi P_n est soit vide soit la trappe maximale incluse dans P .

De plus, la troisième contrainte cherche les trappes qui contiennent le plus grand nombre de places, c-à-d Max-trappe, non marquées.

3. **La condition de démarrage** : Finalement, tous les siphons qui contiennent des trappes non marquées sont détectés. Puis, chaque trappe détectée contient un connecteur qui va être choisi aléatoirement pour envoyer un message vide. Ce choix aléatoire peut être remplacé par un choix manuel de l'utilisateur. Alors, un marquage initial M_{init} est défini pour rendre le sCFN vivant. En effet, M_{init} consiste à initialiser avec (1) les places P_s des composants tel que $\bullet P_s = \{t_s\}$. Aussi, M_{init} initialise les places P_{it} de chaque composant avec (0). Enfin, M_{init} initialise avec un jeton chaque trappe détectée appartenant à un siphon.

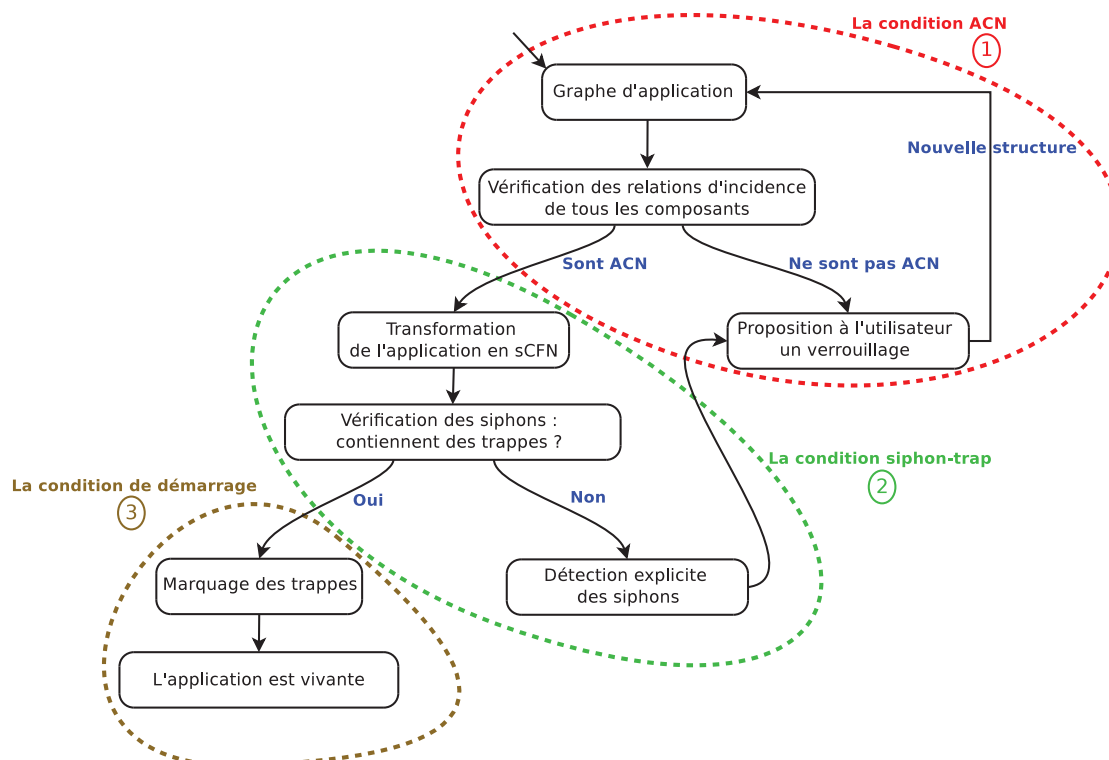


FIGURE 6.16 – Les différentes étapes du processus de démarrage d’une application ComSA.

La figure 6.16 représente les trois étapes du processus de démarrage d’une application ComSA. La première étape la *condition ACN* est représentée par ① contenant les différentes sous-étapes qui servent à vérifier si les relations d’incidence de tous les composants d’une application sont des ACN. La deuxième étape la *condition siphon-trappe* est représentée par ②. Cette étape aide à transformer le graphe d’application en sCFN et à détecter les siphons contenant ou non des trappes. La troisième étape la *condition de démarrage* est représentée par ③ et marque toutes les trappes appartenant à des siphons. Cette dernière étape est réalisable si l’application étudiée ne contient pas de siphons qui ne contiennent pas de trappes.

La table 6.1 montre le résultat après l’application du processus défini par la figure 6.16 sur le graphe d’application de la figure 6.15.

Siphon contenant Trappe	Contenu
1	$[bB_1 : o, s][nbG_2 : o, s][nbG_7 : o, s][S_1 : e, p3d, an, c1]$
2	$[bB_4 : o, s][bB_1 : o, s][nbG_2 : o, s][S_1 : e, p3d, c1, c2]$
3	$[nbG_1 : o, s][I_1 : e, f]$
4	$[sF_1 : i, o][sF_2 : i, o][S_2 : f, en][S_3 : en, f']$
5	$[nbG_8 : o, s][S_3 : e, pos]$
6	$[nbG_6 : o, s][S_3 : e, av]$
7	$[nbB_3 : o, s][sF_1 : i, o][S_3 : e, act, en]$
8	$[sF_2 : i, o][S_2 : e, f]$
9	$[nbG_3 : o, s][nbB_2 : o, s][nbG_5 : o, s][nbG_4 : o, s][V : e, av, act, en, pos]$

TABLE 6.1 – Les siphons contenant une trappe détectés de l'exemple illustré par la figure 6.15

Chaque siphon contenant une trappe obtenu dans notre exemple contient au moins un connecteur. Par exemple, le siphon contenant la trappe numéro 4 se compose de deux connecteurs sF_1 et sF_2 et de deux composants S_2 et S_3 . L'écriture $[sF_1 : i, o][sF_2 : i, o][S_2 : f, en][S_3 : en, f']$ signifie que le siphon contenant la trappe détectée se compose des ports (i, o) du connecteur sF_1 , des ports (i, o) du connecteur sF_2 , des ports (f, en) du composant S_2 et des ports (en, f') du composant S_3 . En effet, un connecteur est choisi aléatoirement pour envoyer un message vide dans le but de garder toujours les siphons détectés marqués et donc l'application est vivante.

6.4 CONCLUSION

Dans ce chapitre, nous avons présenté une approche formelle pour modéliser les applications ComSA. Cette approche est basée sur une classe des réseaux FIFO colorés et des réseaux de Petri colorés nommés les réseaux FIFO colorés stricts. Afin d'analyser le comportement et la structure de ces applications, nous modélisons en sCFN les différents éléments des applications ComSA.

La construction des applications ComSA en assemblant des composants et des connecteurs peut créer des blocages et ces applications peuvent ne pas être vivantes. Dans le but de détecter ces blocages pendant la composition, nous avons proposé un processus d'analyse et de démarrage d'une application ComSA afin de la démarrer correctement et de garantir sa vivacité.

Puisque les applications ComSA sont destinées à la visualisation scientifique interactive et dans le cadre de l'analyse visuelle certaines applications ont besoin d'être reconfigurées dynamiquement. Une telle reconfiguration s'applique soit pour ajouter de nouveaux composants soit pour en retirer. Ainsi, certains composants comme les composants de l'analyse des données ne sont pas persistants et ont besoin d'être lancés ou arrêtés en cours d'exécution. Pour essayer de résoudre ce problème, le chapitre suivant présente un processus de reconfiguration des applications ComSA basé sur la coordination exogène. Cette reconfiguration permet de minimiser le nombre de composants ou de relations d'incidence arrêtés et évite l'arrêt complet de l'application.

RECONFIGURATION DYNAMIQUE DU MODÈLE ComSA

Computer system analysis is like child-rearing ; you can do grievous damage, but you cannot ensure success.

Tom DEMARCO

SOMMAIRE

7.1	INTRODUCTION	123
7.2	LA RECONFIGURATION DYNAMIQUE DES APPLICATIONS ComSA	124
7.2.1	Vue générale du processus de la reconfiguration	124
7.2.2	La délimitation de la région de sécurité	127
7.3	LA CORRECTION DE LA RECONFIGURATION DYNAMIQUE DU MODÈLE ComSA	131
7.3.1	Les invariants de la reconfiguration dynamique du modèle ComSA	131
7.3.2	La vérification de la correction de la reconfiguration dynamique du modèle ComSA	132
7.4	CONCLUSION	136

7.1 INTRODUCTION

Les applications de visualisation scientifique interactives doivent faire face à des évolutions des besoins des utilisateurs. En effet, la modification de la configuration d'une application ComSA est nécessaire pour enrichir ou mettre à jour une application. Par conséquent, la reconfiguration dynamique est nécessaire pour modifier certains composants de l'architecture tandis que l'application est opérationnelle. Dans le contexte de l'analyse visuelle, certaines applications doivent être reconfigurées dynamiquement puisque certains composants, par exemple des étapes d'analyse, ne sont pas persistants et doivent être redémarrés ou arrêtés à la volée.

Dans ce chapitre, nous adressons le problème de la reconfiguration dynamique des applications de visualisation scientifique interactives. Cette reconfiguration consiste à arrêter un ensemble de composants de l'application avant d'interrompre des liens entre certains composants afin de supprimer ou insérer des composants. Nous exposons un algorithme appliqué sur les graphes d'applications pour définir l'ensemble des composants à arrêter. Puis, nous utilisons la sémantique sCFN des applications pour prouver la correction du résultat.

7.2 LA RECONFIGURATION DYNAMIQUE DES APPLICATIONS COMSA

Notre objectif est de reconfigurer dynamiquement une application en cours d'exécution sans l'arrêter complètement. Dans notre contexte, l'utilisateur peut souhaiter :

- l'insertion de nouveaux composants dans le graphe d'application afin d'ajouter, par exemple, de nouvelles étapes de l'analyse sur la simulation en cours ;
- la suppression de composants s'ils ne sont plus utilisés ;
- le remplacement de composants par d'autres.

Notons que ces opérations peuvent être effectuées avec un ensemble de composants qui peuvent également être reliés entre eux. Dans tous les cas, la reconfiguration nécessite de modifier le graphe d'application et de supprimer temporairement des canaux de communication. Le but est alors d'anticiper les conséquences de la suppression d'une communication entre deux ports et de prendre des mesures pour éviter l'arrêt de l'application en cours d'exécution en évitant, par exemple, les débordements (overflows).

7.2.1 Vue générale du processus de la reconfiguration

Le problème de la reconfiguration peut être réduit au problème d'insertion d'un composant entre deux autres. Lors de l'interruption d'un lien entre deux composants, certains ports ne vont plus recevoir de données. Cela peut conduire à des overflows si un composant ne cesse de recevoir de données sur d'autres ports sans être en mesure de commencer une nouvelle itération (aucune relation d'incidence n'est vérifiée).

L'insertion ou la suppression d'un composant signifie changer la coordination exogène de l'application. Par exemple, pour procéder à une insertion d'un composant, certains connecteurs sont retirés afin de supprimer les liens à l'endroit de l'insertion. Ensuite, le nouveau composant est inséré avec de nouveaux connecteurs qui expriment la façon dont il reçoit les données de l'application et la manière dont les résultats sont utilisés. Par conséquent, le processus de reconfiguration doit faire face à la vérification de la correction de la nouvelle application d'une part et la mise en œuvre de l'insertion d'autre part.

La figure 7.1 illustre les différentes étapes de notre processus de reconfiguration et les différents états de l'application au cours de ce processus. Dans cette figure, les *états blancs* signifient que l'application a toujours son architecture initiale c-à-d sa structure avant la reconfiguration et l'*état gris* désigne la nouvelle application chargée à savoir la nouvelle architecture de l'application après la reconfiguration. Les états présentés avec une *bordure pointillée* indiquent que seule la partie non impactée par la reconfiguration est en exécution tandis que les états ayant une *bordure continue* signifient que toute l'application est en exécution.

L'état initial représente l'application en exécution (État 1) qui va être reconfigurée par l'utilisateur. Quand l'utilisateur exprime la requête de la reconfiguration, le nouveau graphe de l'application est construit mais il n'est pas chargé (État 2). Ce nouveau graphe d'application est analysé pour vérifier sa correction. Dans ce contexte, nous

faisons appel aux techniques définies dans le chapitre 6 afin de détecter les blocages et définir la condition de démarrage pour garantir la vivacité de l'application. Si la nouvelle application n'est pas valide, la reconfiguration est refusée et l'application reste non modifiée. De plus, ces techniques peuvent fournir des suggestions pour corriger l'application, par exemple, verrouiller certains comportements de composants comme défini dans la section 6.3.

Si la nouvelle application est valide (État 3), le graphe d'application en question est analysé afin de définir l'implémentation de la reconfiguration. Cette implémentation consiste à définir un moyen sûr pour enlever les connecteurs liés à l'insertion ou à la suppression de composants, car ces suppressions peuvent générer des overflows.

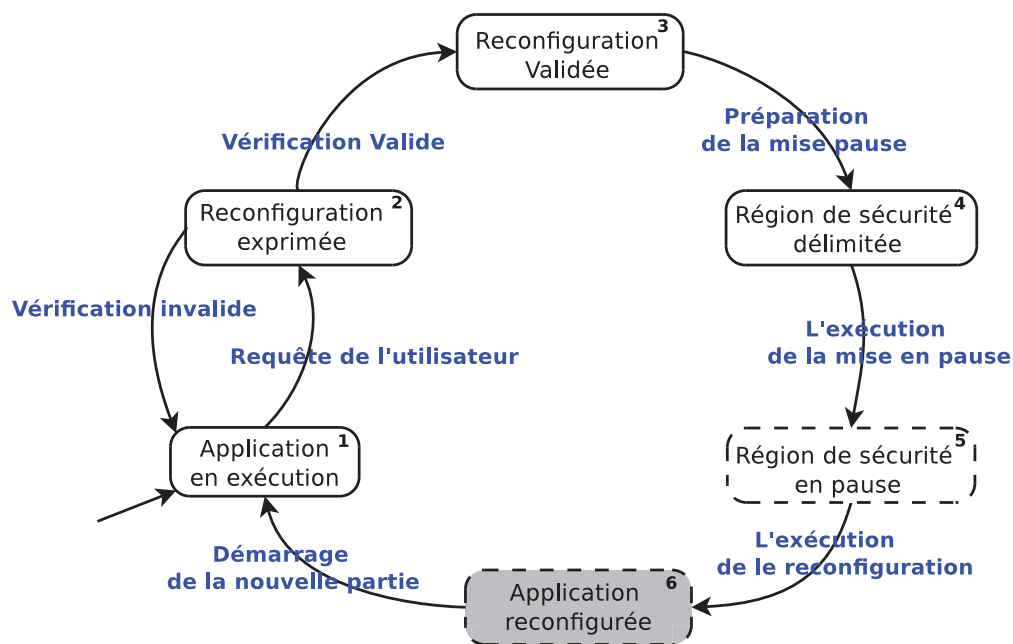


FIGURE 7.1 – Les différents états de la reconfiguration.

En effet, certains composants peuvent ensuite être bloqués par un manque de données sur un port d'entrée alors que les autres ports continuent à recevoir des données. Pour éviter ces problèmes lors d'une reconfiguration, une partie de l'application doit être mise en pause avant la suppression d'un connecteur. C'est l'objectif de l'algorithme 3.

La région de sécurité délimitée obtenue en (État 4) est illustrée par la figure 7.3. Le rectangle avec une bordure pointillée est la région de sécurité. Les composants qui sont à l'extérieur de ce rectangle sont toujours en exécution alors que les composants qui sont à l'intérieur sont en pause. Certains composants peuvent être en pause partiellement, par exemple les composants *A* et *B*. En se basant sur cette description, la mise en pause est implémentée (État 5). Le résultat de l'algorithme 3 indique comment réaliser cette mise en pause. Jusqu'à cette étape, l'application initiale est toujours en

Algorithme 3 : Algorithme de la mise en pause d'une application ComSA

Entrées : $App = (Comp \cup Conn, DI \cup TI)$ et c le connecteur supprimé tel que $C_1^o \xrightarrow{c} C_2^i$

Sorties : $UConn$ l'ensemble de connecteurs à neutraliser,
 $PComp$ l'ensemble de composants à mettre en pause,
 RDI un ensemble de liaisons de données à supprimer

// L'étape initiale illustrée par la figure 7.2(a)

```

1  $UConn := \emptyset; PComp := \emptyset;$ 
2  $SIR := \{r \in IR_{C_2} \text{ s.t. } C_2^i \in IR^{in}(r)\};$ 
  // l'ensemble des relations d'incidence arrêtées
3  $SPorts := IR^{in}(SIR) \setminus IR^{in}(IR_{C_2} \setminus SIR);$ 
  // l'ensemble de ports qui doivent être affamés
4  $Done := \emptyset;$ 
5 Tant que  $SPorts \neq \emptyset$  faire
  // L'étape montante illustrée par la figure 7.2(b)
6   Soit  $C_n^i \in SPorts;$ 
7    $Done := Done \cup \{C_n^i\};$ 
8   Soit  $C_{n-1}^o \xrightarrow{c} C_n^i \subseteq DI;$ 
9   Si  $t_c \in \{bBuffer, nbBuffer, sFifo\}$  Alors
10    Si  $\exists r \in IR^{in}(C_{n-1}^o) \text{ s.t. } IR^{in}(r) = \emptyset$  Alors
11       $PComp := PComp \cup \{C_{n-1}\};$ 
12       $SIR := SIR \cup IR_{C_{n-1}};$ 
13    Sinon
14       $SIR := SIR \cup \{IR^{in}(C_{n-1}^o)\};$ 
15       $SPorts := SPorts \cup (IR^{in}(SIR \cap IR_{C_{n-1}}) \setminus IR^{in}(IR_{C_{n-1}} \setminus SIR));$ 
16    Sinon
17       $UConn := UConn \cup \{c\};$ 
  // L'étape descendante illustrée par la figure 7.2(c)
18   $SOut := (IR^{out}(IR^{out}(C_n^i))) \setminus (IR^{out}(IR_{C_n} \setminus IR^{out}(C_n^i)));$ 
19   $SPorts := SPorts \cup_{p_o \in SOut} \{p_i | \exists p_o \xrightarrow{c} p_i \subseteq DI \text{ s.t. } t_c \text{ blocking}\};$ 
20   $SPorts := SPorts \setminus Done;$ 
21  $RDI := resolvesFIFOCycle(App, SIR);$ 

```

exécution (les *états blancs*). Après, seule la partie de l'application non impactée par la reconfiguration reste en exécution (les états avec une bordure pointillée). La reconfiguration est ensuite effectuée pour implémenter le nouveau graphe d'application (État 6). Les composants à changer sont supprimés et les nouveaux sont ajoutés. Enfin, le démarrage de l'application reconfigurée est effectué pour revenir à un état de fonctionnement normal (État 1).

7.2.2 La délimitation de la région de sécurité

Le point de départ de la délimitation de la région de sécurité est un ensemble de connecteurs qui vont être retirés pour effectuer une telle reconfiguration.

L'algorithme 3 prend en entrée un graphe d'application et renvoie un ensemble de connecteurs à neutraliser c-à-d les empêcher livrer des messages, un ensemble de composants qu'il faudra pour réaliser explicitement une pause ainsi qu'un ensemble de liens de données afin d'isoler la région de la sécurité par rapport au reste de l'application. Cet algorithme est présenté et exécuté avec un seul connecteur à enlever, mais il peut facilement être généralisé pour un ensemble de connecteurs.

Intuitivement, quand un canal de communication $C_1^o \xrightarrow{c} C_2^i$ doit être coupé entre deux composants C_1 et C_2 , une ou plusieurs relations d'incidence du composant C_2 peuvent être affamées à cause du manque de données. Cela peut provoquer des overflows sur les ports d'entrée liés à ces relations d'incidence quand ils reçoivent encore de nouvelles données. Pour éviter ce problème, toutes les relations d'incidence susceptibles de fournir des données aux ports de relations d'incidence affamées doivent être arrêtées. Il existe deux façons d'arrêter une relation d'incidence. Si elle a des ports d'entrée, il suffit de les affamer. Sinon, si elle n'a pas de ports d'entrée, un composant spécial appelé *pauseC* va être relié au port s pour que toutes les relations d'incidence du composant en question sont arrêtées.

La façon d'affamer un port d'entrée p dépend du type du connecteur c relié à ce port. Si le connecteur est de type *Greedy*, il suffit d'enlever le lien de déclenchement relié au port s de ce connecteur afin de l'empêcher d'envoyer des nouvelles données. Si ce connecteur est de type *sFIFO* ou *Buffer*, les relations d'incidence du port de sortie qui lui fournit des données, doivent être arrêtées. Finalement, si le graphe d'application contient des cycles *sFIFO*, certaines liaisons de données dans ces cycles doivent être supprimées.

Plus précisément, l'algorithme 3 détecte d'abord les relations d'incidence qui vont être arrêtées dans le composant C_2 (ligne 2) et déduit l'ensemble des ports d'entrée du composant C_2 à affamer pour éviter l'overflow (ligne 3). Cet ensemble contient juste les ports d'entrée des relations d'incidence arrêtées.

Ensuite, la boucle générale de l'algorithme 3 traite chaque port à affamer (l'ensemble $SPorts$). Durant l'itération de la boucle, de nouveaux ports peuvent être ajoutés à l'ensemble $SPorts$. L'ensemble *Done* aide à éviter de traiter plusieurs fois le même port. Un port p de l'ensemble $SPorts$ est traité en deux étapes : une *étape montante* et une *étape descendante*.

- L'étape montante dépend d'abord du connecteur c lié au port p , ensuite, elle dépend de la relation d'incidence fournissant les données au connecteur c .
 - Si le connecteur est sans perte, c-à-d de type *Buffer* ou *sFIFO*, il existe deux cas :
 1. Si au moins une des relations d'incidence qui fournit des données au connecteur c qui n'a pas de ports d'entrée, pour éviter les overflows il

Algorithme 4 : L'algorithme pour résoudre les cycles sFIFO

Entrées : $App = (Comp \cup Conn, Dl \cup Tl)$ un graphe d'application et SIR un ensemble des relations d'incidence

Sorties : $RemovedsFifo$ un ensemble des liens

```

1  $RemovedsFifo := \emptyset;$ 
2 Pour chaque  $c \in Conn$  s.t.  $type_c \neq sFifo$  Faire
3    $\lfloor Remove(c);$ 
4 Pour chaque  $C \in Comp$  s.t.  $\nexists r \in SIR \cap IR_C$  Faire
5    $\lfloor Remove(C);$ 
6 Tant que  $App \neq \emptyset$  faire
7    $Continue := true;$ 
8   Tant que  $Continue$  faire
9      $Continue := false;$ 
10    Pour chaque  $C \in Comp$  Faire
11       $T_{in} := \forall C^i \in Pin^C \nexists \langle x, C^i \rangle \in Dl;$ 
12       $T_{out} := \forall C^o \in Pout^C \nexists \langle C^o, y \rangle \in Dl;$ 
13      Si  $T_{in} \vee T_{out}$  Alors
14         $\lfloor Remove(C);$ 
15         $\lfloor Continue := true;$ 
16    Pour chaque  $c \in Conn$  Faire
17      Si  $\nexists \langle c^i, x \rangle \in Dl \vee \nexists \langle y, c^i \rangle \in Dl$  Alors
18         $\lfloor Remove(c);$ 
19         $\lfloor Continue := true;$ 
20  Si  $App \neq \emptyset$  Alors
21    Soit  $\langle x, y^i \rangle \in Dl$  où  $y$  est un connecteur ;
22     $Dl := Dl \setminus \{\langle x, y \rangle\};$ 
23     $RemovedsFifo := RemovedsFifo \cup \{\langle x, y \rangle\};$ 

```

suffit d'arrêter le composant producteur en reliant son port de déclenchement s au composant $pauseC$. Ce composant est ajouté à l'ensemble $PComp$ et ses relations d'incidence sont ajoutées à l'ensemble SIR des relations d'incidence qui vont être arrêtées (ligne 11 et ligne 12).

2. Sinon, l'ensemble des relations d'incidence fournissant des données au connecteur c sont ajoutées à l'ensemble SIR (ligne 14).

Dans les deux cas, l'ensemble des ports d'entrée du composant producteur qui doivent être affamés, sont ajoutés à l'ensemble $SPorts$. Cet ensemble se compose de l'ensemble des ports d'entrée du composant qui ont juste les ports d'entrée des relations d'incidence qui vont être arrêtées (ligne 15).

- Si le connecteur c est de type *Greedy*, il est facile d'enlever son lien de déclenchement. En effet, si un composant en cours d'exécution fournit à ce connecteur des données, le connecteur les ignore pour éviter les overflows. Donc, ce connecteur est ajouté à $UConn$ (ligne 17).
- L'étape descendante recherche les ports qui ne délivrent plus de données. Ces ports sont les ports de sortie des relations d'incidence arrêtées (ligne 18) représentés par l'ensemble $SOut$. L'ensemble des ports d'entrée qui sont connectés à l'un de ces ports avec un canal de communication géré par un connecteur bloquant, ne vont plus recevoir de données. Par conséquent, ils doivent être ajoutés à l'ensemble $SPorts$ des ports affamés (ligne 19). La ligne 20 de l'algorithme 3 évite tout simplement de traiter plusieurs fois le même port et garantit la fin de l'algorithme.

Une fois que tous les ports affamés sont traités, l'algorithme 3 identifie l'ensemble des connecteurs à neutraliser et l'ensemble des composants à mettre en pause à l'aide du composant $pauseC$.

Dans le cas où le graphe d'application contient un cycle ne contenant que des connecteurs de type *sFIFO* et si ce cycle intersecte la partie de l'application à arrêter, il faut enlever un lien de données appartenant à ce cycle afin d'arrêter complètement la partie de l'application en question. Cette gestion de cycles est l'objectif de la fonction *resolvesFIFOCycle* (ligne 21) définie par l'algorithme 4.

La figure 7.2 illustre les différentes étapes de la mise en pause d'une application ComSA réalisées par l'algorithme 3.

La figure 7.2(a) montre l'étape initiale du processus de cet algorithme. Dans cette figure, les ports i et i' du composant C_2 sont ajoutés à l'ensemble $SPorts$ tandis que le port i'' ne va pas être ajouté car c'est un port d'entrée de la relation d'incidence r_3 c-à-d $i'' \in RI^{in}(r_3)$ et cette relation d'incidence n'est pas arrêtée.

La figure 7.2(b) illustre l'étape montante permettant d'explorer les chemins contenant que des connecteurs sans perte. Dans cet exemple, le port o du composant C_{n-1} fournit des données au connecteur c . Il est alimenté par deux relations d'incidence r_1 et r_2 qui doivent être arrêtées, ce qui n'est pas le cas pour la relation d'incidence r_3 . Les deux ports i et i' doivent être affamés et sont ajoutés à l'ensemble $SPorts$, par contre le port i'' ne va pas être affamé. En effet, i'' est un port d'entrée pour les deux relations d'incidence r_2 et r_3 . Cette dernière n'a pas besoin d'être arrêtée, de telle sorte que le composant sera en mesure de traiter des données reçues via le port i'' .

La figure 7.2(c) schématise l'étape descendante inspectant les chemins ne contenant que des connecteurs bloquants. Dans cette figure, le port o du composant C_n est un port de sortie de la relation d'incidence arrêtée r_2 , mais aussi de r_1 qui reste toujours en exécution et fournit des données à ce port qui ne s'ajoute pas à $SOut$. Le port o' est uniquement alimenté par les relations d'incidence arrêtées de telle sorte qu'il appartient à $SOut$. Dans notre exemple, le port i du composant C_{n+1} sera affamé si le port o du composant C_n ne produit plus de données.

À partir d'une application $App = (Comp \cup Conn, DI \cup TI)$ et d'un connecteur c à enlever, l'algorithme 3 retourne les ensembles $UConn$, $PComp$ et RDI pour procéder

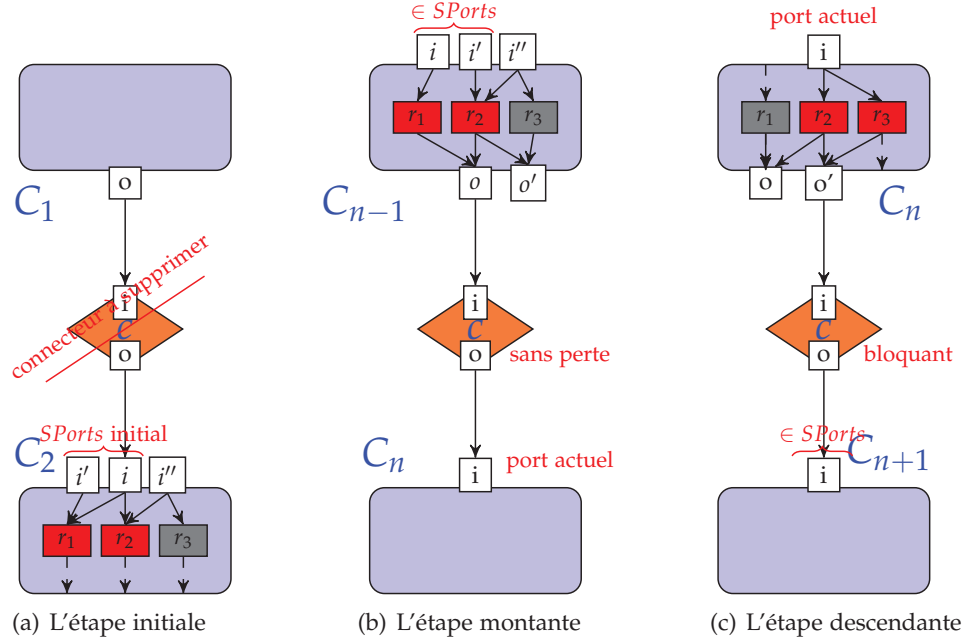


FIGURE 7.2 – L'illustration des différentes étapes de l'algorithme 3

à la reconfiguration. Ces ensembles sont utilisés pour implémenter les actions pour mettre en pause la partie impactée par cette reconfiguration. Le résultat est un graphe d'application défini comme suit :

$$\tilde{App}_c = (Comp \cup \{pauseC\} \cup Conn, \tilde{DI} \cup \tilde{TI}),$$

avec $\tilde{DI} = DI \setminus RDI$ et $\tilde{TI} = TI \setminus \{\langle x^e, c^s \rangle | c \in UConn\}$.

Dans le graphe \tilde{App}_c , la partie arrêtée A_s de l'application est isolée par rapport à la partie A_r représentant le reste de l'application qui reste opérationnel, cela est illustré par la figure 7.3. Dans l'étape descendante de l'algorithme, les relations d'incidence des composants de la partie A_r envoient des données vers A_s à l'aide des connecteurs du type *Greedy* où les liens de déclenchement sont enlevés (la bordure ①). Par conséquent, ces données sont simplement ignorées par ces connecteurs. L'ignorance de ces données est garantie, au niveau de l'étape descendante, à l'aide des relations d'incidence des composants de la partie A_r qui peuvent recevoir des données émises par la partie A_s qui contient des connecteurs non bloquants (la bordure ②). Ainsi, les relations d'incidence de cette bordure reçoivent des données vides tant que la mise en pause est maintenue. À l'intérieur de la partie A_s , tous les composants avec au moins une relation d'incidence sans ports d'entrée sont mis en pause à l'aide du composant *pauseC*, les autres ne peuvent pas recevoir de données depuis A_r et A_s qui ne contiennent pas de cycles *SFIFO*. Cela signifie que le nombre de données circulant dans cette partie diminue et toutes les relations d'incidence vont s'arrêter à cause de

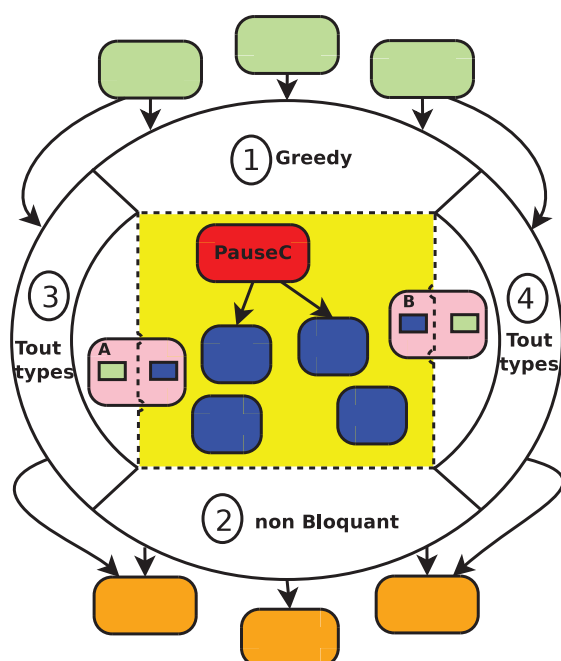


FIGURE 7.3 – La région de sécurité impactée par la reconfiguration.

l'absence de données. Quand cela arrive, la reconfiguration peut être effectivement appliquée pour construire la nouvelle application.

7.3 LA CORRECTION DE LA RECONFIGURATION DYNAMIQUE DU MODÈLE COMSA

7.3.1 Les invariants de la reconfiguration dynamique du modèle ComSA

Le modèle ComSA a un ensemble d'invariants pour assurer la correction (soundness) des architectures des applications de visualisation scientifique interactives. Ces invariants sont des détails structurels de l'architecture d'une application qui restent stables avant et après une reconfiguration dynamique. Les invariants ComSA sont utilisés pour évaluer les architectures des applications de visualisation scientifique interactive. Les invariants ComSA sont :

- **la connectivité des composants ou/et des connecteurs** : cet invariant vérifie que tous les composants ou/et les connecteurs ComSA ne sont pas isolés c-à-d le graphe d'application construit est bien un graphe connexe ;
- **l'unicité de la connexion des ports de données** : cet invariant assure que chaque port d'entrée de données ne peut être alimenté que par un seul lien ;
- **l'existence des relations d'incidence correctes** : cet invariant vérifie qu'au moins une relation d'incidence d'un composant doit avoir soit tous ses ports connectés à au moins un lien chacun, soit aucun de ses ports connectés ;

- **la satisfaction de contrainte du composant** : cet invariant contrôle que chaque composant est un ACN c-à-d ses relations d'incidence sont sous forme d'un réseau ACN (définition 27) ;
- **la vérification de la structure de l'application** : cet invariant garantit qu'une application ComSA est bien formée c-à-d tous ses composants sont bien connectés en entrée et en sortie (définition 12).

Une architecture d'une application ComSA est structurellement cohérente si les cinq invariants précédemment définis sont satisfaits. Nous garantissons que chaque architecture construite et modifiée par une reconfiguration dynamique du modèle ComSA est cohérente.

L'application illustrée par la figure 6.15 vérifie bien les cinq invariants de l'approche ComSA. Supposons que l'utilisateur souhaite la reconfiguration de cette application suite à la suppression du connecteur nbG_8 reliant le port pos du composant S_2 et le port pos du composant S_3 . Cette suppression va être effectuée pour insérer un nouveau composant effectuant des calculs supplémentaires sur les positions des atomes. Le résultat est d'arrêter tous les composants impliqués dans les calculs de force. En appliquant l'algorithme 3 de la mise en pause sur cette application ComSA, nous obtenons la région de sécurité définie et illustrée par la figure 7.4 où les connecteurs neutralisés et les relations d'incidence arrêtées sont représentés par des couleurs transparentes. Le composant I_1 est le seul composant appartenant à l'ensemble $PComp$ des composants à arrêter explicitement. Dans ce cas, nous utilisons le composant $PauseC$ en le connectant avec le composant I_1 au niveau du port de déclenchement s .

En effet, les relations d'incidence arrêtées sont :

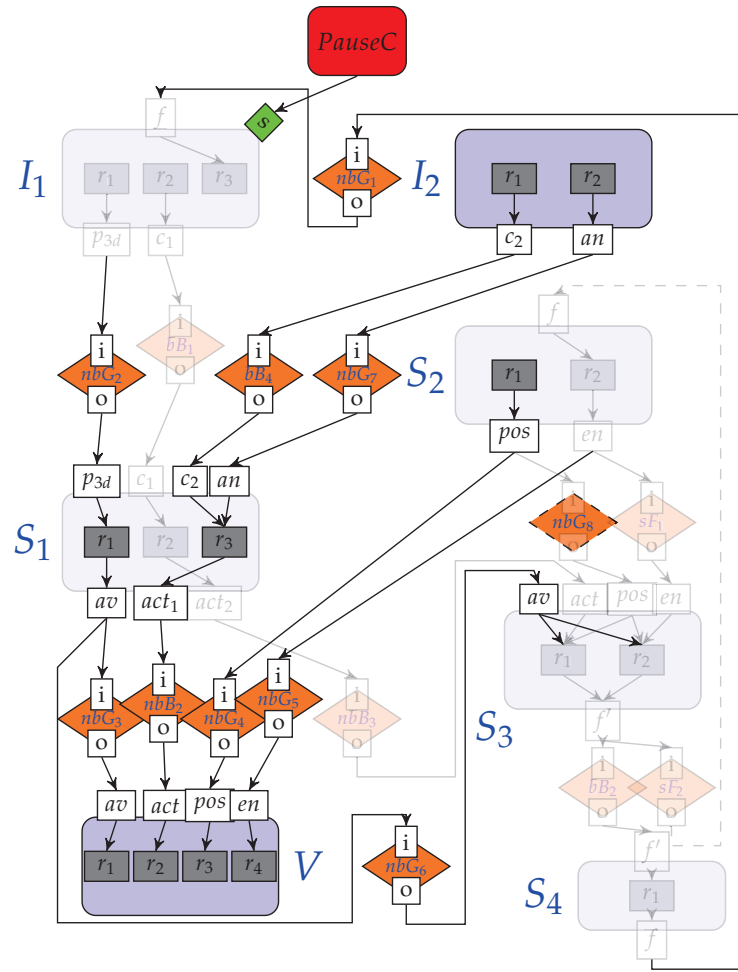
- r_1, r_2 et r_3 du composant I_1 ;
- r_2 du composant S_1 ;
- r_2 du composant S_2 ;
- r_1 et r_2 du composant S_3 ;
- r_1 du composant S_4 .

Et les connecteurs neutralisés sont :

- bB_1, nbB_3, sF_1, bB_2 et sF_2 .

7.3.2 La vérification de la correction de la reconfiguration dynamique du modèle ComSA

Dans le graphe obtenu $\tilde{App}_c = (Comp \cup \{pauseC\} \cup Conn, \tilde{DI} \cup \tilde{TI})$ où \tilde{DI} et \tilde{TI} sont les ensembles des liens obtenus en appliquant l'algorithme 3 sur le graphe d'application $App = (Comp \cup Conn, DI \cup TI)$ et où le connecteur c est enlevé. Les autres résultats de l'algorithme 3 sont notés $UConn_{\tilde{App}_c}$, $PComp_{\tilde{App}_c}$ and $RDI_{\tilde{App}_c}$. De plus, un port appartenant à \tilde{App}_c est appelé *port affamé* s'il est un port d'une relation d'incidence appartenant à $RDI_{\tilde{App}_c}$ ou s'il est un port d'un connecteur connecté à une relation d'incidence de $RDI_{\tilde{App}_c}$.

FIGURE 7.4 – La région de sécurité impactée par la suppression du connecteur nbG_8 de la figure 6.15.

Maintenant, nous allons prouver que toutes les relations d'incidence de l'ensemble $RDI_{\tilde{App}_c}$ sont arrêtées dans le graphe \tilde{App}_c et que cet arrêt ne causera pas d'overflow et n'empêchera pas d'autres relations d'incidence de s'exécuter.

Lemme 4 Soient $App = (Comp \cup Conn, DI \cup TI)$ et c le connecteur à enlever en modifiant App . Soit x^p un port d'une relation d'incidence de $RDI_{\tilde{App}_c}$ et y^q un port d'une relation d'incidence n'appartenant pas à $RDI_{\tilde{App}_c}$. Tout chemin de données dans \tilde{App}_c à partir de y^q jusqu'à x^p passe par z^0 où $z \in UConn_{\tilde{App}_c}$ et $type_z \in \{nbGreedy, bGreedy\}$ et tout chemin de données dans \tilde{App}_c à partir de x^p jusqu'à y^q passe par z^i où $z \notin UConn_{\tilde{App}_c}$ et $type_z \in \{nbGreedy, nbBuffer\}$.

Démonstration. Les deux parties du lemme 4 sont des conséquences directes de l'algorithme 3. En effet, tout port $x^{p'}$ tel qu'il existe un chemin de données, à partir de $x^{p'}$

jusqu'à x^p , qui ne passe pas par un connecteur *Greedy* est donc un port affamé et toutes les relations d'incidence sont impliquées dans $RDI_{A\tilde{p}p_c}$. Symétriquement, tout port $x'^{p'}$ tel qu'il existe un chemin de données, à partir de x^p jusqu'à $x'^{p'}$, qui ne passe pas par un connecteur non bloquant est un port affamé, cela conduit à la même conclusion. \square

En conséquence, les relations d'incidence arrêtées de $RDI_{A\tilde{p}p_c}$ ne produisent pas d'overflow. En effet, puisque toutes les données entrantes de la partie en cours d'exécution de l'application entrent par un connecteur *Greedy* pour arrêter cette partie, tous les messages non utilisés seront simplement jetés. De même, les relations d'incidence arrêtées de $RDI_{A\tilde{p}p_c}$ ne provoquent pas la famine des autres relations d'incidence puisque les données venant de $RDI_{A\tilde{p}p_c}$ passent par un connecteur non bloquant ce qui signifie que les consommateurs de ces données recevront des messages vides lorsque le producteur est arrêté. Enfin, nous devons prouver que tous les éléments de $RDI_{A\tilde{p}p_c}$ seront effectivement arrêtés au niveau de $A\tilde{p}p_c$.

Lemme 5 Soient $App = (Comp \cup Conn, DI \cup TI)$ et c le connecteur à enlever en modifiant App . Alors, toutes les relations d'incidence de $RDI_{A\tilde{p}p_c}$ seront arrêtées.

Avant de prouver le lemme 5, nous allons donner une condition suffisante pour montrer qu'un réseau sCFN n'est pas vivant. Cette condition est présentée sous forme de la définition 35 et du théorème 4.

Définition 35 Soit S un sCFN. Une place p de S est appelée place décroissante si elle est une place source ou une place de sortie d'une seule transition décroissante. Une transition t de S est appelée transition décroissante si au moins une de ses places d'entrée est une place décroissante.

Dans le sCFN de la figure 6.2, P_1 est une place source donc une place décroissante. t_1 est une transition décroissante puisque la place P_1 est une place source. Ainsi, P_2 est une place décroissante car c'est une place de sortie de t_1 . La transition t_2 est donc une transition décroissante et P_3 est une place décroissante selon la définition 35.

Le théorème 4 affirme que les transitions décroissantes peuvent ne pas être toujours vivantes. Par conséquent, toutes ces transitions vont mourir.

Théorème 4 Soient S un sCFN et M un marquage de S . Alors, toutes les transitions décroissantes peuvent être appliquées qu'un nombre limité de fois.

Pour démontrer ce théorème, nous allons construire un ordre sur les places décroissantes basé sur leurs dépendances comme suit :

- $L_0 = \{p | p \text{ est une place source}\}$
- $L_{i+1} = \{p | p \text{ est une place décroissante, } p \notin L_0 \cup \dots \cup L_i \text{ et } \exists t \in T \text{ et } p' \in L_i \text{ tel que } \langle p', t \rangle \in A \text{ et } \langle t, p \rangle \in A\}$

Dans le sCFN de la figure 6.2(a), $L_0 = \{P_1\}$, $L_1 = \{P_2\}$ et $L_3 = \{P_3\}$ et $L_i = \emptyset$ quand $i > 3$.

Pour le sCFN S , nous définissons $depth(S)$ un nombre entier représentant le pro-

fondeur tel que $L_{depth(S)} = \emptyset$ et $L_{depth(S)-1} \neq \emptyset$. Si $L_0 = \emptyset$, $depth(S)$ est 0. La profondeur de sCFN de la figure 6.2(a) est 4.

Avant de prouver le théorème 4, nous allons présenter les remarques suivantes :

1. Par définition, des nouveaux jetons peuvent arriver dans une place décroissante seulement en appliquant une transition décroissante ;
2. L'ensemble $\bigcup_{i \geq 0} L_i$ est exactement l'ensemble des places décroissantes par construction ;
3. Par construction aussi, pour chaque transition décroissante t , au moins une de ses places décroissantes d'entrée a un rang strictement inférieur à toutes les places décroissantes de sortie de t .

Finalement, la démonstration du théorème 4 utilise la notion de *rang* pour définir un ordre lexicographique sur un tuple de nombres entiers naturels. Ainsi, nous définissons d'abord le tuple $F_M(S) = \langle k_0, \dots, k_n \rangle$ pour un marquage M du sCFN S où $k_i = \sum_{p \in L_i} |p|_M$. Alors, si nous considérons l'ordre $<$ dans les tuples, nous prouvons que pour chaque marquage M , le franchissement de la transition décroissante t donne un marquage M' tel que $F_M(S) < F_{M'}(S)$.

Démonstration. (du théorème 4) Soient S un sCFN et M un marquage tel que la transition t est franchie. Nous considérons le marquage obtenu M' à l'aide du franchissement de la transition t .

- Si t n'est pas une transition décroissante, nous avons $F_M(S) \leq F_{M'}(S)$ puisque, depuis la première remarque ci-dessus, une telle transition n'ajoute pas de nouveaux jetons dans les places décroissantes ;
- Si t est une transition décroissante, nous avons $F_M(S) < F_{M'}(S)$. En effet, à partir de la troisième remarque ci-dessus, comme t est une transition décroissante et au moins une de ses places d'entrée a un rang i strictement inférieur au rang d'une place de sortie de t . Cela signifie que $F_M(S) = \langle k_0, \dots, k_n \rangle$ et $F_{M'}(S) = \langle k'_0, \dots, k'_n \rangle$ tel que $k_j = k'_j$ pour $j < i$ et $k_i > k'_i$, donc, $F_M(S) > F_{M'}(S)$.

Puisqu'il n'y a pas une séquence infinie décroissante de tuples de nombres naturels, les transitions décroissantes ne peuvent être appliquées qu'un nombre fini de fois, donc ces transitions ne sont pas vivantes. \square

Démonstration. (du lemme 5) Soit $r \in RDI_{App_c}$, soit C le composant possédant la relation d'incidence r .

Si $RI^{in}(r) = \emptyset$, alors le port s de C est connecté avec le composant $pauseC$. Le sCFN $_C(pauseC)$, c-à-d la modélisation du composant $pauseC$ en sCFN, est réduit à une seule place nommée P_{pause} . Dans le sCFN(App_c) la place P_{pause} représentant $pauseC$ est une place source, ainsi la transition t_s de C est une transition décroissante car P_{pause} appartient à l'ensemble de ses places d'entrée. Alors, la place P_s de C est une place décroissante. Toutes les transitions de sCFN $_C(C)$ ont P_s comme place d'entrée, elles sont toutes des transitions décroissantes ainsi que les places représentant les ports de sortie.

Nous allons traiter le cas où la relation d'incidence r n'a pas de ports d'entrée. Si $\exists C^i \in RI^{in}(r)$ qui n'est pas connecté, alors la place représentant C^i dans $sCFN_G(\tilde{App}_c)$ est une place source et toutes les transitions qui ont cette place comme place d'entrée sont des transitions décroissantes.

Si $\exists C^i \in RI^{in}(r)$ et $c' \in Conn$ tel que $\langle c'^o, C^i \rangle \in \tilde{DI}$ et $type_{c'} \neq sFIFO$, en se basant sur l'algorithme 3 nous savons que $c' \in UConn_{\tilde{App}_c}$ ce qui signifie que le port s de c' est non connecté, ainsi, la place P_s qui représente s dans $sCFN_G(\tilde{App}_c)$ est une place source. Donc, la transition t_{out} (et t_s si elle existe) est une transition décroissante, voir les figures 6.7, 6.8 et 6.9 qui montrent que la place P_o est une place décroissante. La transition t_{C^i} est alors une transition décroissante aussi bien que la place P_{C^i} .

Si tous les ports d'entrée de r sont connectés avec des connecteurs $sFIFO$, nous ne pouvons pas conclure directement. Cependant nous remarquons que si $\exists C^i \in RI^{in}(r)$ tel que P_{C^i} est une place décroissante, alors, nous pouvons tirer la même conclusion que dans les deux cas précédents. En appliquant les deux algorithmes 3 et 4, nous savons qu'il n'existe pas de cycle dans \tilde{App}_c qui utilise uniquement des composants et connecteurs $sFIFO$, ce qui signifie que pour chaque port d'entrée C^i de C il existe un chemin qui commence par un composant qui correspond à l'un des quatre cas précédents. Donc, la place correspondante au port d'entrée C^i de ce composant est une place décroissante.

Nous avons prouvé que toutes les transitions qui représentent les éléments de $RDI_{\tilde{App}_c}$ dans $sCFN_G(\tilde{App}_c)$ sont décroissantes. En effet et en se basant sur le théorème 4, ces transitions peuvent être appliquées seulement un nombre fini de fois ce qui signifie qu'elles vont être arrêtées. \square

L'absence d'overflow est prouvé par le lemme 4. Par conséquent, l'algorithme 3 permet de réaliser une reconfiguration dynamique basée sur les comportements, représentés par les relations d'incidence, guidés par les données et la coordination exogène d'une application ComSA. Cet algorithme met en pause la partie impactée par la modification de l'architecture de l'application pour minimiser le nombre des services arrêtés fournis par les différents composants et évite l'arrêt complet de l'application.

7.4 CONCLUSION

Dans ce chapitre, nous avons proposé une technique de reconfiguration pour les applications ComSA basée sur la coordination exogène favorisée par le modèle ComSA. Cette technique isole la partie de l'application impactée par la reconfiguration par rapport aux autres parties non impactées. Étant donné que des schémas de communication du modèle ComSA contiennent des buffers, des débordements peuvent apparaître lors d'une reconfiguration si les composants récepteurs consommant les données de ces buffers ne sont pas opérationnels. Dans le but de détecter ces débordements, nous avons proposé une reconfiguration persistante et préventive de ce problème de buffers.

La reconfiguration des applications de visualisation scientifique interactives proposée est appliquée quand par exemple un utilisateur souhaite ajouter ou supprimer un composant selon ses besoins. Alors, notre approche garantit une reconfiguration correcte et sécurisée, c-à-d l'application ne s'arrête pas complètement pendant la reconfiguration et fonctionnera correctement après l'application des résultats de notre travail concernant la détection de blocages et la vivacité présentées dans le chapitre 6. Nous avons présenté des algorithmes pour mettre en œuvre les différents aspects du fonctionnement de la reconfiguration des applications ComSA.

Tout les aspects concernant le modèle ComSA notamment (1) la présentation de ses différents éléments présentés dans le chapitre 5, (2) la modélisation des applications ComSA basée sur les réseaux FIFO colorés stricts et la détection de blocages présentées dans le chapitre 6, (3) la reconfiguration dynamique présentée dans ce chapitre, seront illustrés et expérimentés dans le chapitre suivant.

VALIDATION ET MISE EN ŒUVRE

No amount of experimentation can ever prove me right ; a single experiment can prove me wrong.

Albert EINSTEIN

SOMMAIRE

8.1	INTRODUCTION	139
8.2	LA PLATEFORME COMSATool	139
8.3	ARCHITECTURE DE COMSATool	142
8.3.1	Le vérificateur de la vivacité en sCFN	143
8.3.2	Le reconfigurateur dynamique des applications ComSA	143
8.4	LES DÉTAILS D'IMPLANTATION DE COMSATool	146
8.4.1	Implantation du Vérificateur	146
8.4.2	Implantation du Reconfigurateur	150
8.5	CONCLUSION	151

8.1 INTRODUCTION

Dans ce chapitre, nous nous intéressons aux expérimentations pour valider notre travail. Pour ce faire, nous avons implanté les algorithmes décrits dans les chapitres précédents. Ce chapitre illustre les différents aspects de la mise en œuvre. La section 8.2 détaille les fonctionnalités de la plateforme développée. Dans la section 8.3, nous présentons les éléments conceptuels de l'architecture proposée. La section 8.4 expose les détails techniques de sa mise en œuvre.

8.2 LA PLATEFORME COMSATool

La plateforme ComSATool est une implémentation de nos algorithmes de construction des applications de visualisation scientifique interactives basées sur l'approche ComSA. Ces algorithmes permettent de modéliser les applications ComSA, vérifier leur vivacité, comme décrit dans le chapitre 6, et les reconfigurer dynamiquement, comme décrit dans le chapitre 7. La plateforme ComSATool peut être utilisée par des utilisateurs informaticiens ou non informaticiens. Il existe deux modes d'utilisation de ComSATool concernant l'affichage des applications à construire. Le premier mode utilise les graphes d'applications pour la modélisation. Le second mode permet d'afficher le réseau FIFO coloré strict. Les différentes fonctionnalités de ComSATool sont :

- la construction des applications ComSA basées sur des composants et des connecteurs, en les ajoutant ou en les supprimant ;
- la vérification que les applications ComSA sont bien formées, comme décrit dans la section 5.3 ;
- la génération du réseau FIFO coloré strict de l'application ComSA créée ;
- la vérification de la vivacité des applications ComSA, avec une vérification que les relations d'incidences respectent la condition ACN, comme défini dans la section 6.3 ;
- la détection des siphons, pour aider l'utilisateur à modifier son application ;
- la détection des siphons contenant des trappes ;
- la possibilité de définir le marquage initial ;
- la simulation du réseau sCFN généré, soit en utilisant le marquage initial, soit en ajoutant ou en supprimant des jetons ;
- la reconfiguration des applications ComSA en supprimant ou en ajoutant des composants.

La fenêtre principale du ComSATool est illustrée par la figure 8.1. Cette fenêtre ne traite que les graphes d'applications ComSA.

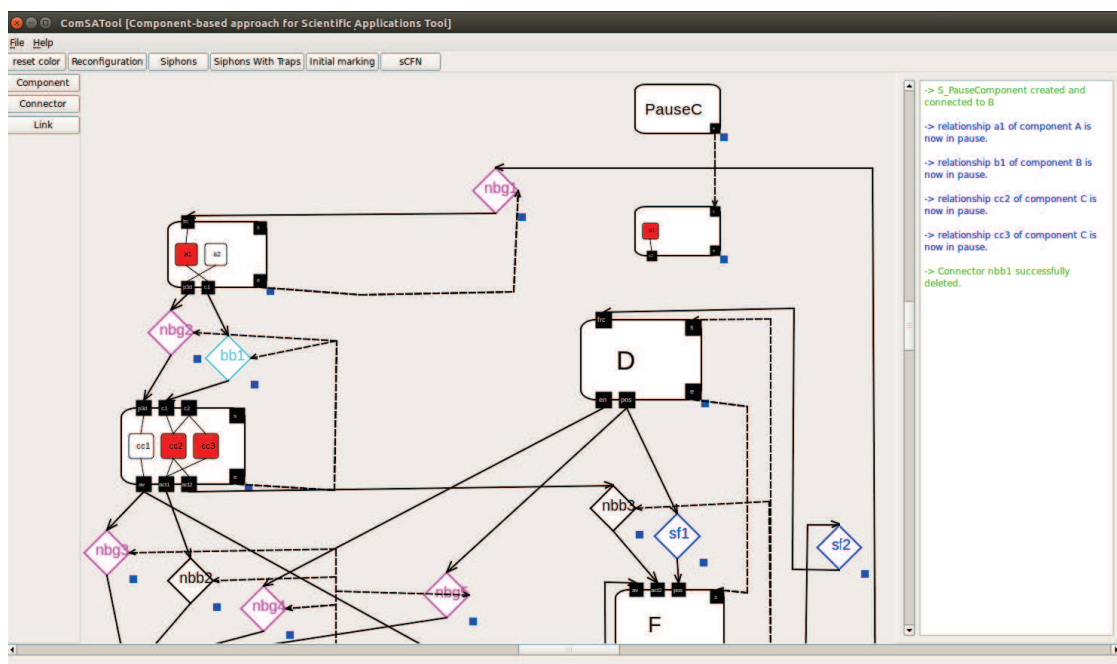


FIGURE 8.1 – L'interface graphique principale de la plateforme ComSATool.

Pour pouvoir construire une application ComSA, l'utilisateur a besoin de créer une nouvelle application via le menu *File* → *New* et utiliser les trois boutons : *Component*, *Connector* et *Link* qui se trouvent en haut à gauche. Le bouton *Component* permet de créer les composants en créant leurs ports d'entrée et de sortie et leurs relations d'incidences. Le bouton *Connector* permet de créer les connecteurs et choisir leurs types. Le

bouton *Link* permet d'avoir les liens de donnée et de déclenchement afin d'assembler les composants en les connectant avec les connecteurs. L'utilisateur peut enregistrer le schéma de son application créée via le menu *File* → *Save* et l'importer ultérieurement via le menu *File* → *Open*.

Dans la même fenêtre, l'utilisateur peut reconfigurer dynamiquement son application ComSA en utilisant le bouton *Reconfiguration*. Cette reconfiguration permet à l'utilisateur de supprimer un composant. Après une telle suppression ComSATool modifie la couleur, en rouge, des relations d'incidence qui sont en pause.

Dans la même fenêtre, l'utilisateur peut afficher les siphons, à l'aide du bouton *Siphons*, pour qu'il puisse verrouiller certaines relations d'incidence des composants qui ne respectent pas la condition ACN. Ces siphons sont affichés dans le panneau droit. De plus et en utilisant le bouton *Siphons With Traps*, nous pouvons visualiser dans le panneau droit les siphons contenant une trappe pour définir la condition de démarrage en insérant automatiquement des messages vides qui vont garantir la vivacité de l'application ComSA construite. Ces messages vides représentent le marquage initial.

La plateforme ComSATool permet d'interagir avec les réseaux FIFO colorés stricts. Ce mode de fonctionnement s'effectue en utilisant le bouton *sCFN* de la fenêtre principale. La figure 8.2 illustre le sCFN généré de l'application ComSA de la figure 8.1.

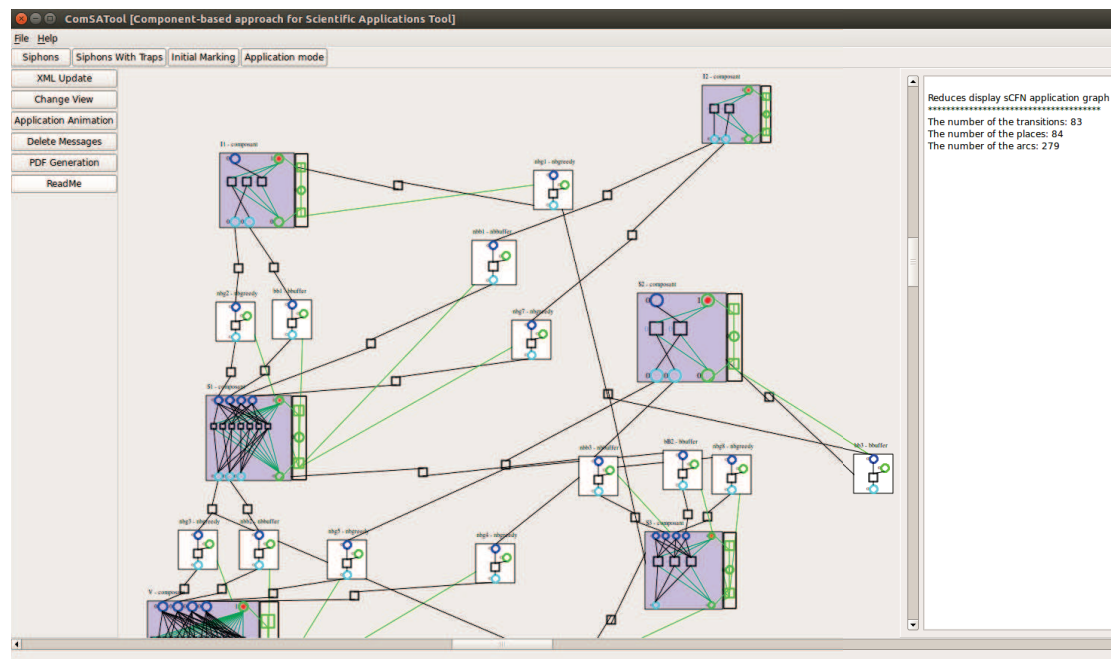


FIGURE 8.2 – L'interface graphique des sCFN de la plateforme ComSATool.

Au niveau de cette fenêtre, l'utilisateur a la possibilité d'afficher le sCFN de l'application ComSA construite. Lorsque le sCFN est généré, l'utilisateur peut afficher les siphons et les siphons contenant des trappes. Par contre dans ce mode, ces derniers sont affichés en utilisant la notion de place du sCFN. De plus, pour chaque sCFN

généralisé un marquage initial peut se définir et s'afficher à l'aide du bouton *Initial Marking*. Dans le même mode, l'utilisateur peut réaliser une simulation du sCFN avec une animation en temps réel des jetons au niveau des places. L'interface permet d'ajouter et de supprimer des jetons manuellement avec la possibilité de générer le sCFN sous format PDF.

Notre plateforme ComSATool se compose de deux éléments principaux :

1. le module *Vérificateur* ;
2. le module *Reconfigureur*.

8.3 ARCHITECTURE DE COMSATool

L'architecture globale du ComSATool est illustrée par la figure 8.3.

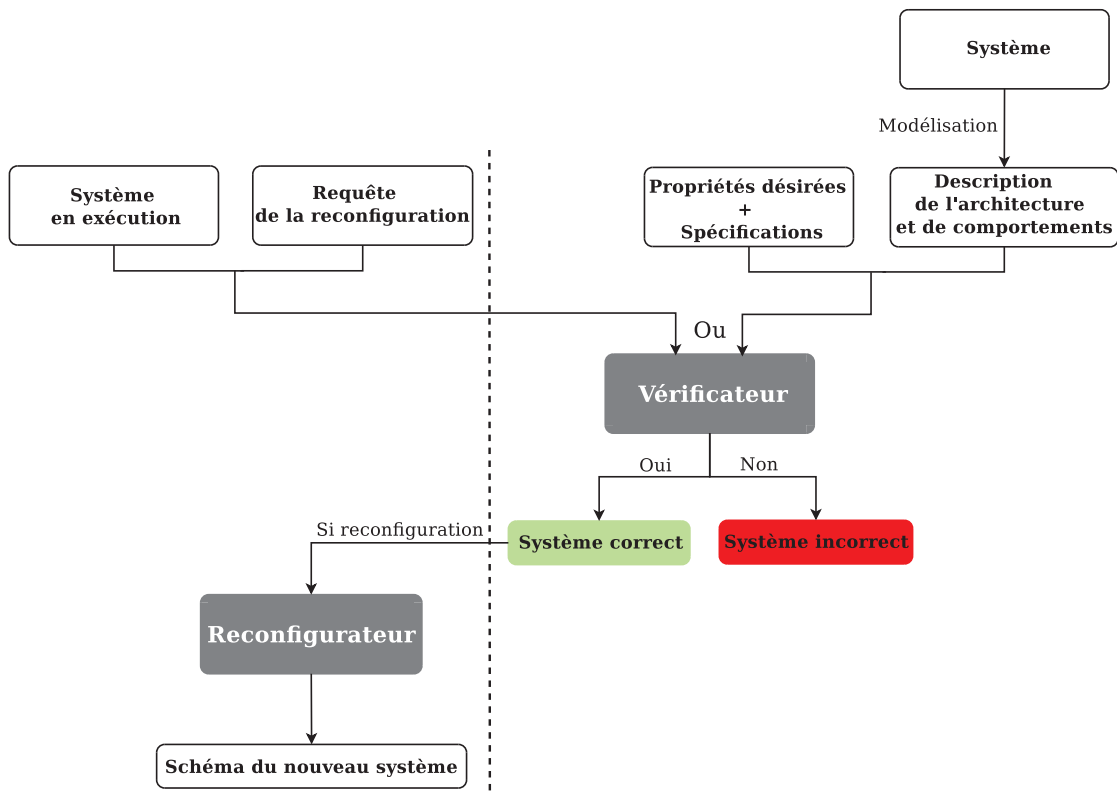


FIGURE 8.3 – L'architecture de la plateforme ComSATool.

Cette architecture est basée sur le module *Vérificateur* et le module *Reconfigureur* qui représentent le cœur de notre plateforme. Ces deux modules s'appuient essentiellement sur les travaux abordés dans cette thèse dans le cadre du projet *ExaviZ*. Le module *Vérificateur* implante les algorithmes pour modéliser et transformer le graphe d'une application ComSA en sCFN. De plus, ce module intègre la définition de la

configuration de démarrage des applications ComSA impliquant la garantie de leur vivacité présentée dans le chapitre 6. En ce qui concerne le deuxième module, c-à-d le *Reconfigureur*, il implante les algorithmes de la reconfiguration dynamique des applications ComSA présentés dans le chapitre 7.

8.3.1 Le vérificateur de la vivacité en sCFN

La figure 8.4 illustre l'architecture globale du module *Vérificateur* que nous proposons. Ce module se base sur trois sous-modules :

1. *ComSA2sCFN* ;
2. *sCFN2SAT* ;
3. *Solveur SAT*.

Le sous-module *ComSA2sCFN* implémente les étapes et les algorithmes décrits dans la section 6.2. Il a comme entrée le graphe d'une application de visualisation scientifique interactive et comme sortie le sCFN correspondant. En effet, son but est de transformer le graphe d'application en réseau FIFO coloré strict représentant une modélisation formelle qui préserve la sémantique et le comportement des applications ComSA.

La modélisation en sCFN fournie par *ComSA2sCFN* représente l'entrée du sous-module *sCFN2SAT*. Ce dernier produit une traduction vers un modèle booléen SAT en utilisant une procédure SAT itérée à l'aide des formules de contraintes décrites dans la section 6.3. Ce sous-module contient deux sous-modules, le premier *SiphonTrapSAT* est chargé de réaliser la traduction SAT pour identifier les siphons contenant des trappes.

Par contre, le deuxième sous-module *SiphonSAT* sert à effectuer la traduction pour chercher les siphons sans trappe. Ces traductions générées par le sous-module *sCFN2SAT* sont interprétées en utilisant le sous-module *Solveur SAT* pour vérifier l'existence des siphons contenant des trappes et des siphons sans trappe. Ce dernier est basé sur le solveur open-source *MINISAT* [187] qui a été modifié pour qu'il puisse générer toutes les solutions possibles.

Si ce module produit des solutions qui représentent les siphons ne contenant pas de trappe, alors le système est incorrect. En effet, le module *Simulateur* prend en entrée ces solutions pour produire des propositions de corrections à l'utilisateur. Ces propositions sont fournies dans le but de changer la structure de l'application pour qu'elle ne contienne plus de siphons représentant les blocages, comme expliqué dans la section 6.3. De plus, le sous-module *Solveur SAT* est utilisé pour chercher les siphons contenant des trappes afin de définir un marquage initial.

8.3.2 Le reconfigureur dynamique des applications ComSA

Le deuxième module de notre plateforme est illustré par la figure 8.5. Ce module dispose de quatre sous-modules :

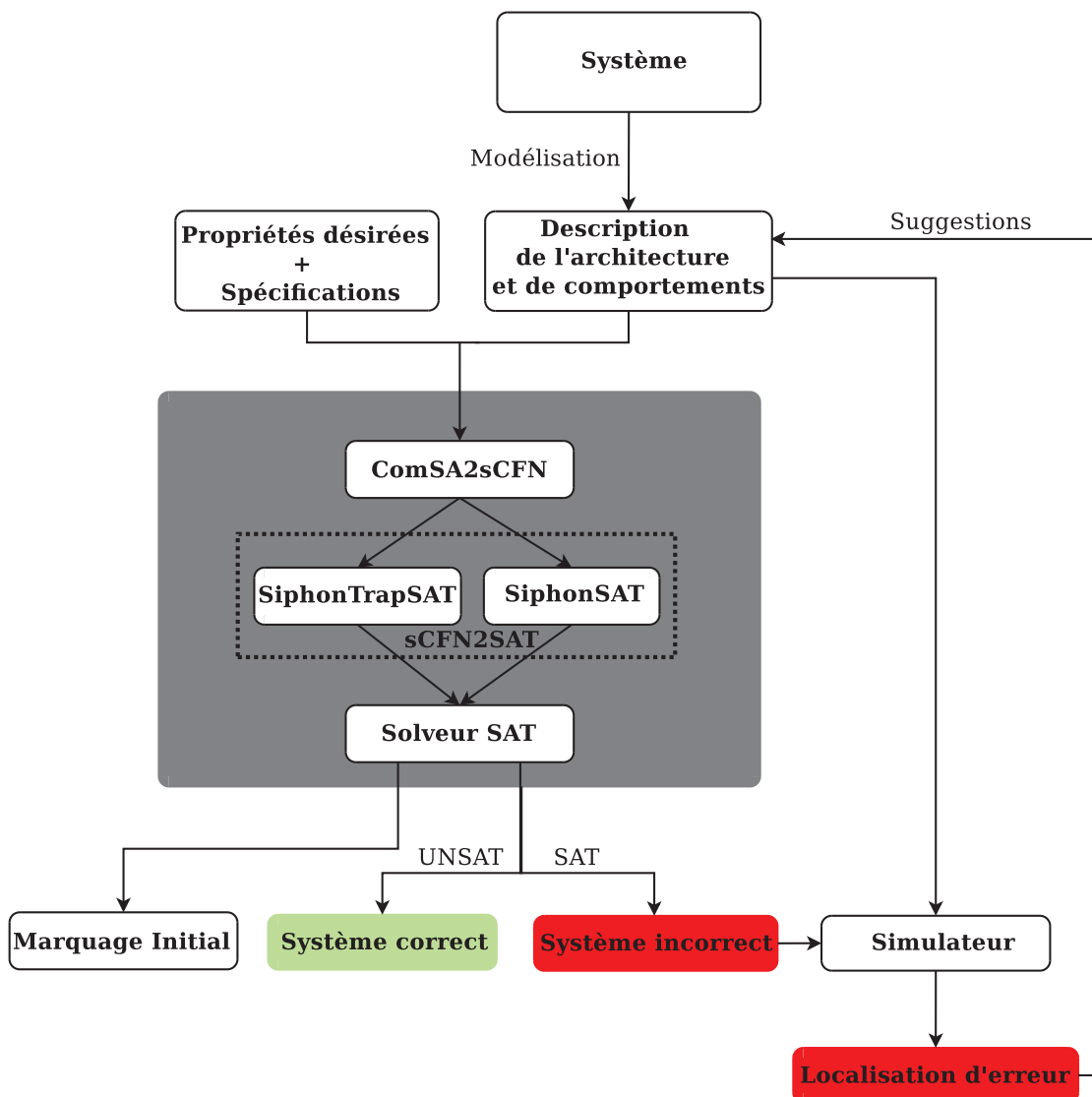


FIGURE 8.4 – L'architecture du vérificateur de la plateforme ComSA.

1. *ComSAUPReconfig* (ComSA up step reconfiguration) ;
2. *ComSADOWNReconfig* (ComSA down step reconfiguration) ;
3. *ComSAResolveCycle* (ComSA resolve cycles) ;
4. *ComSA2sCFN*.

Avant que le module *Reconfigurateur* intervienne, ce module fait appel au module *Vérificateur* pour vérifier que la nouvelle architecture souhaitée ne va pas contenir de blocages et que la nouvelle application est bien formée.

le module *Reconfigurateur* est une implantation de l'algorithme 3 représentée par trois étapes implantées par les sous-modules : *ComSAUPReconfig*, *ComSADOWNRecon-*

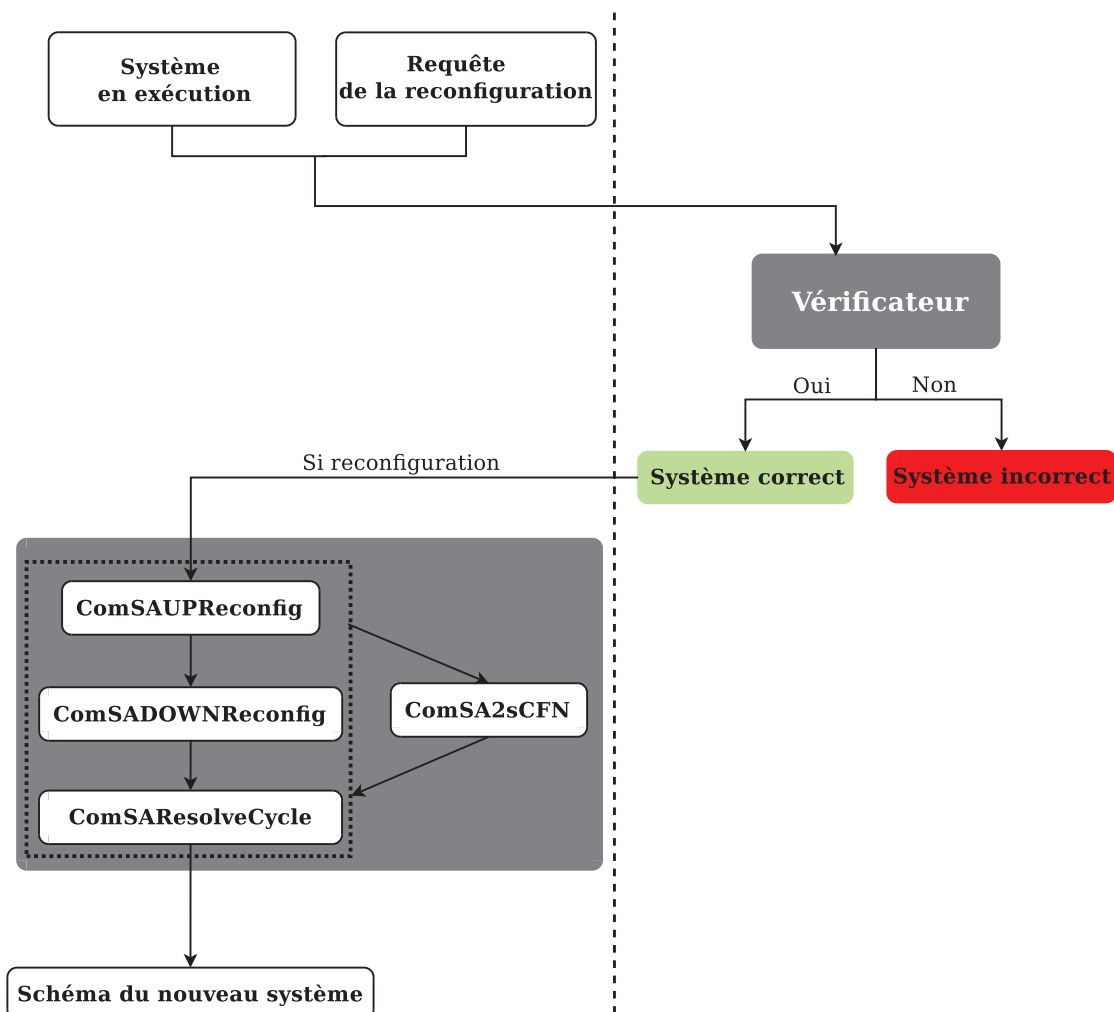


FIGURE 8.5 – L'architecture du reconfigurateur de la plateforme ComSA.

fig et *ComSAResolveCycle*. En effet, le point d'entrée du *Reconfigurateur* est le schéma d'une application en exécution ainsi qu'une requête d'utilisateur d'une reconfiguration souhaitée. Le sous-module *ComSAUPReconfig* implante la procédure illustrée par la figure 7.2(b) pour effectuer l'étape montante de la délimitation de la région de sécurité pour reconfigurer une telle application. Quand ce sous-module finit ses traitements, le sous-module *ComSADOWNReconfig* prend le relais dans le but de finir en descendant la délimitation de la région de sécurité. Ce sous-module implante la procédure représentée par la figure 7.2(c). De plus, quand ce dernier sous-module génère ses résultats, le sous-module *ComSAResolveCycle* intervient pour résoudre les cycles *sFIFO* dans le graphe d'application en implantant l'algorithme 4.

Après la délimitation finale de la région de la sécurité pour une reconfiguration donnée, le sous-module *ComSA2sCFN* peut être utilisé pour modéliser la partie de l'application qui est mise en pause.

Dans la suite, nous présentons les détails techniques de la mise en œuvre des différents modules de notre plateforme ComSATool.

8.4 LES DÉTAILS D'IMPLANTATION DE COMSATTOOL

Cette section décrit d'abord la spécification d'entrées de la plateforme ComSATool concernant la description d'un graphe d'application. Ensuite, le module *Vérificateur* est présenté en détails. Finalement, le module *Reconfigureur* qui se charge de la reconfiguration dynamique est exposé dont le but de réaliser une reconfiguration correcte et sécurisée. L'entrée principale du ComSATool est un graphe d'application. Nous avons utilisé le langage de description XML pour la représentation de la description de l'architecture et de comportements. Un graphe d'application est un fichier XML qui a comme racine l'élément `<application>`. Cet élément regroupe un ensemble d'éléments définis par l'annexe A.

8.4.1 Implantation du Vérificateur

Le *Vérificateur* de notre plateforme ComSATool a été implémenté en JAVA sous forme d'un module qui peut être utilisé comme une boîte noire. Ce module prend en entrée un fichier XML décrivant le graphe d'application ComSA. Ce fichier représente le fichier d'entrée du sous-module *ComSA2sCFN* qui a été implémenté sous forme d'un parseur XML et développé en utilisant l'API JDOM (Java Document Object Model). Ce sous-module a comme sortie deux fichiers. Le premier a comme extension *.scfn* et décrit le réseau FIFO coloré strict qui modélise le graphe d'application d'entrée. Le deuxième fichier a comme extension *.pdf* contenant le sCFN de l'application. La figure 8.6 montre le sCFN généré par le sous-module *ComSA2sCFN* de l'exemple illustré par la figure 6.15.

Chaque ligne du fichier *.scfn* décrivant le sCFN correspond aux places d'entrée ou de sortie d'une transition. Si une transition a des places d'entrée et des places de sortie, elle sera définie par deux lignes. La structure d'une ligne est :

- Si la transition a des places d'entrée, alors elle est représentée par la structure :
 T_nom de la transition nom de l'objet contenant cette transition :pre :P_nom du port_nom de l'objet contenant cette place ;
- Si la transition a des places de sortie, alors elle est représentée par la structure :
 T_nom de la transition nom de l'objet contenant cette transition :post :P_nom du port_nom de l'objet contenant cette place.

Dans les deux structures, T symbolise une transition et P symbolise une place. De plus, le nom de l'objet peut être le nom d'un composant ou d'un connecteur. La figure 8.7 montre un exemple d'un fichier généré par le sous-module *ComSA2sCFN*.

Le fichier généré par le sous-module *ComSA2sCFN* est utilisé comme entrée pour les deux sous-modules *SiphonTrapSAT* ou *SiphonSAT*. Le premier est implémenté pour

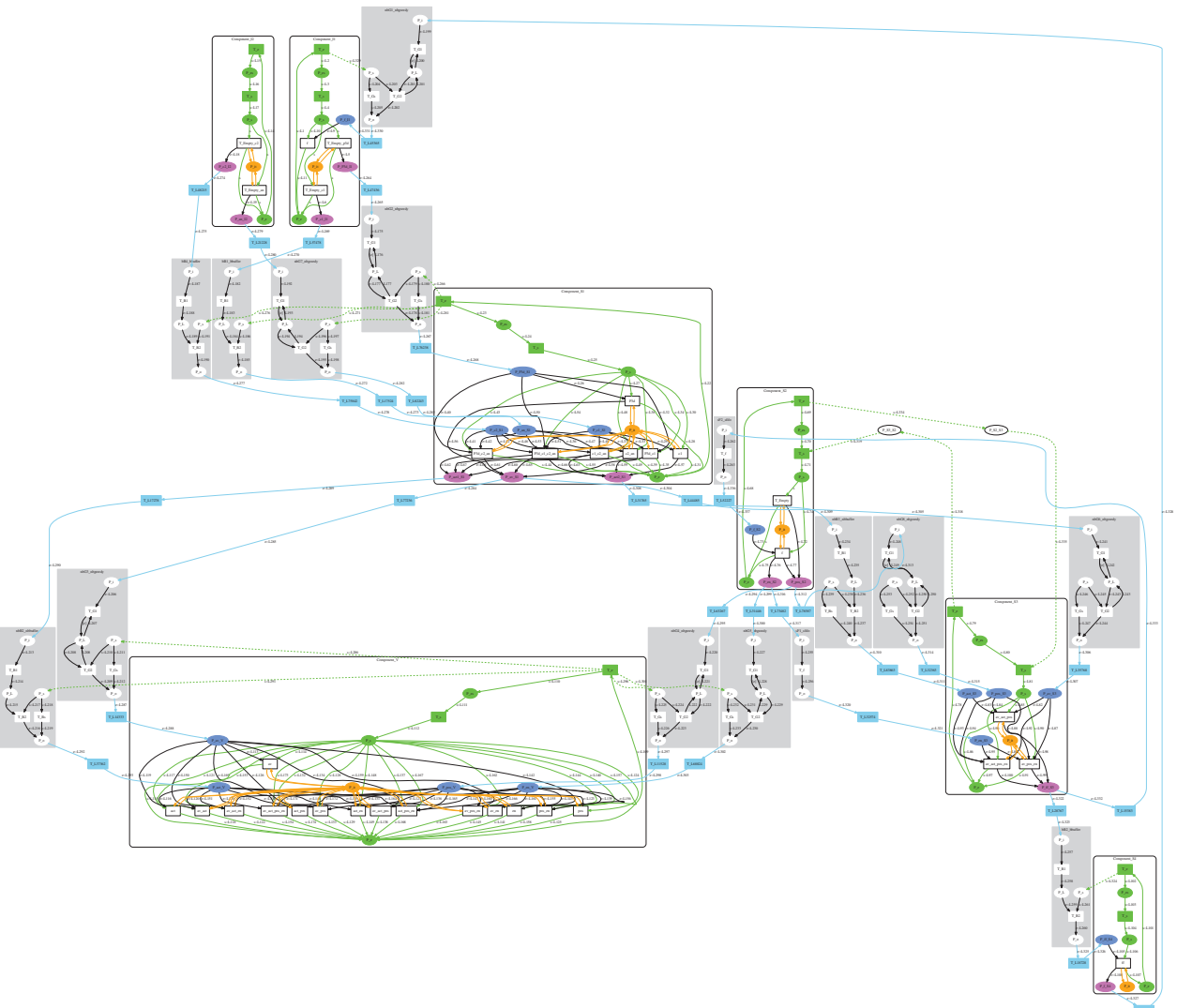


FIGURE 8.6 – *Le sCFN du graphe d'application de l'exemple de la figure 6.15.*

chercher les siphons contenant des trappes et le deuxième est développé pour détecter les siphons. Ces deux sous-modules sont implémentés en utilisant les contraintes définies dans la section 6.3.2. Ces contraintes sont traduites vers un codage généré et représenté sous un fichier ayant comme extension *.dimacs*. Le listing B.1 représente un extrait de ce fichier concernant l'application de la figure 6.15. Chaque ligne de ce

fichier est une liste de variables séparées par des espaces et terminée par un o. Cette liste représente une clause qui est une disjonction de littéraux et un littéral est soit une variable x positive, soit sa négation $\neg x$.

```

T_t'_{i_1}_C : pre : P_{l_1}_C1
T_t'_{i_1}_C : post : P_{i_1}_C
T_t'_{i_2}_C : pre : P_{l_2}_C2
T_t'_{i_2}_C : post : P_{i_2}_C
T_{t_{i_1}i_2}_C : pre : P_{i_1}_C : P_{i_2}_C
T_{t_{i_1}i_2}_C : post : P_{o_1}_C
T_{t_{i_2}}_C : pre : P_{i_2}_C
T_{t_{i_2}}_C : post : P_{o_1}_C
T_{t_s}_C : pre : P_{P_s}_C
T_{t_s}_C : post : P_{P_{es}}_C
T_{t_e}_C : pre : P_{P_{es}}_C
T_{t_e}_C : post : P_{P_e}_C
T_{t'_{o_1}}_C : pre : P_{o_1}_C
T_{t'_{o_1}}_C : post : P_{l_3}_C3
T_{t'_{o_2}}_C : pre : P_{o_2}_C
T_{t'_{o_2}}_C : post : P_{l_4}_C4

```

FIGURE 8.7 – Le fichier *.scfn* définissant le sCFN de la figure 6.6.

Dans un fichier *.dimacs*, une variable est représentée par un entier compris entre 1 et n et sa négation \neg est représentée par le signe -. En effet, les lignes d'un fichier *.dimacs* représentent la conjonction des clauses du problème. La première ligne d'un fichier *.dimacs* s'écrit sous le format suivant :

$$p \quad cnf \quad n \quad c$$

avec n le nombre de variables qui représentent les places et c le nombre de clauses du problème. La figure 8.8 contient un exemple d'un fichier *.dimacs*

```

p   cnf   4   3
1   -3    0
2    3   -1    0
2    1   -4    0

```

FIGURE 8.8 – Un fichier *.dimacs* exemple avec 4 variables et 3 clauses.

Le fichier *.dimacs* généré par le sous-module *SiphonTrapSAT* ou le sous-module *SiphonSAT* est ensuite introduit dans le sous-module *Solveur SAT* qui va vérifier l'existence d'un siphon ou d'un siphon contenant une trappe. Si ce sous-module prend en entrée le fichier *.dimacs* généré par le sous-module *SiphonSAT*, alors il va générer toutes les solutions représentant les siphons de l'application étudiée. Ces modèles vont être les entrées du module *Simulateur* qui va donner la correspondance entre les siphons générés et les éléments du graphe d'application de l'entrée initiale du module *Véri-*

ificateur. Sinon, le sous-module génère *UNSAT* et donc le système est correct c-à-d le système ne contient pas de siphons.

Si ce sous-module prend en entrée le fichier *.dimacs* généré par le sous-module *SiphonTrapSAT*, alors il va générer toutes les solutions représentant les siphons contenant des trappes. Dans ce cas, le module *Vérificateur* définit un marquage initial.

Afin d'évaluer notre approche et particulièrement le module *Vérificateur*, nous avons mené des tests sur plusieurs exemples d'applications ComSA. Ces expérimentations ont été effectuées sur une machine de 2,10 GHz CPU et 16 Go de mémoire. Notre objectif est d'étudier la vérification des applications ComSA en variant le nombre de composants ce qui implique de varier le nombre de transitions, de places et d'arcs. La table 8.1 représente les statistiques obtenues pour la recherche des siphons et celle qui consiste à rechercher les siphons contenant des trappes. Dans cette table la colonne *propriétés* permet de mentionner quelle propriété est étudiée à savoir : siphons contenant des trappes ($\text{Trappes} \in \text{Siphons}$) ou siphons. Les colonnes $|P|$, $|T|$ et $|A|$ permettent de donner respectivement le nombre des places, des transitions et des arcs générés au niveau du sCFN. La colonne $|\text{Composants}|$ représente le nombre de composants appartenant à l'application étudiée. La colonne *Temps d'exécution* représente le temps de traitement des propriétés.

	propriétés	$ P $	$ T $	$ A $	Composants	Temps d'exécution
App_A	Trappes \in Siphons	240	12	32	2	0.63 sec
	Siphons	15	12	32		0.04 sec
App_B	Trappes \in Siphons	1332	45	121	4	12.6 sec
	Siphons	36	45	121		0.13 sec
App_C	Trappes \in Siphons	2070	53	141	5	29.9 sec
	Siphons	45	53	141		0.19 sec
App_D	Trappes \in Siphons	2970	62	162	7	58.23 sec
	Siphons	54	62	162		0.25 sec
App_E	Trappes \in Siphons	3540	67	173	8	82.49 sec
	Siphons	59	67	173		0.3 sec
App_F	Trappes \in Siphons	6642	85	217	11	290.29 sec
	Siphons	81	85	217		0.32 sec
App_G	Trappes \in Siphons	16002	138	359	16	1629.62 sec
	Siphons	126	138	359		0.39 sec

TABLE 8.1 – L'évaluation de la performance du module Vérificateur de l'approche ComSA.

Afin de calculer les propriétés recherchées, nous avons utilisé un solveur SAT pour énumérer les solutions des formules propositionnelles issues de l'encodage de la sous-section 6.3.2 et qui ont été transformées au format CNF afin de permettre l'utilisation de tels solveurs. La figure 8.9 représente l'évolution du temps nécessaire pour énumérer toutes les solutions en fonction du nombre de places de départ. Comme on peut l'observer, plus le nombre de places augmente plus le temps nécessaire au calcul des siphons contenant des trappes augmente. Ceci s'explique par la taille de l'encodage

puisque le nombre de places n générées pour chercher cette propriété est quadratique par rapport au nombre de places n' générées pour chercher les siphons c-à-d $n = n'(n' + 1)$.

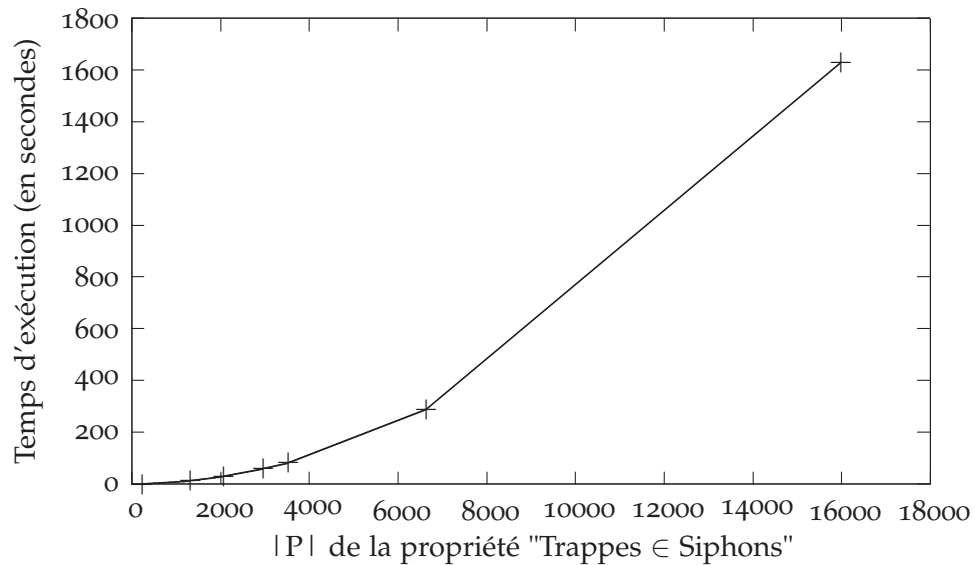


FIGURE 8.9 – L'évaluation de la performance de l'étude de la propriété "Trappes ∈ Siphons" par rapport au nombre de places.

8.4.2 Implantation du Reconfigurateur

Les sous-modules du module *Reconfigurateur* ont également été implémentés en JAVA. Ce module prend en entrée le fichier XML décrivant le graphe d'application en exécution et la requête de la reconfiguration définie par l'utilisateur. Avant l'intervention de ce module, le module *Vérificateur* vérifie que le nouveau graphe d'application est correct. S'il est correct, le module *Reconfigurateur* dirige les données d'entrée qui sont le graphe d'application existant et la requête d'utilisateur vers le sous-module *ComSAUPReconfig* qui se charge de définir la région de sécurité impactée par la reconfiguration la partie ① de la figure 7.3. Ce sous-module met en pause les relations d'incidence impactées en positionnant l'attribut `ispaused` du fichier de description du graphe d'application à `true`, comme illustré par exemple par le listing A.1 de l'annexe A. De plus, le sous-module *ComSADOWNReconfig* permet d'identifier la partie ② de la même figure, en modifiant également l'attribut `ispaused` des relations d'incidence impactées par la reconfiguration dynamique à `true`. Le sous-module *ComSAResolveCycle* prend en entrée les résultats des deux derniers sous-modules et résout les cycles *sFIFO*. En effet, le module *Reconfigurateur* retourne le nouveau schéma du nouveau système.

8.5 CONCLUSION

Dans ce chapitre, nous avons mis en pratique notre approche ComSA proposée dans ce manuscrit. Ceci a montré la faisabilité de notre plateforme avec différentes applications ComSA.

Nous avons présenté les différents éléments de la plateforme ComSATool sous forme de deux modules principaux :

- Le *Vérificateur* effectue l'analyse des entrées concernant la structure d'architecture d'application et les propriétés désirées. Il vérifie la cohérence des applications par rapport aux invariants ComSA, la vivacité de ces applications en les transformant en réseaux FIFO colorés stricts et en utilisant des traductions vers le problème de décision SAT. Ces traductions permettent de détecter les siphons et les siphons contenant des trappes ;
- Le *Reconfigureur* permet d'effectuer une reconfiguration dynamique sécurisée sans arrêter toute l'application c-à-d arrêter juste les éléments impactés par une telle opération de la reconfiguration ComSA. Ce module contient plusieurs sous-modules permettant la réalisation d'une reconfiguration souhaitée.

Actuellement, les efforts de développement se focalisent sur la modélisation des applications ComSA en sCFN et exactement sur la réduction des réseaux sCFN générés. Cette réduction permet d'obtenir un sCFN réduit avec un nombre réduit de places et un nombre réduit de transitions en préservant la vivacité. Cela nous permettra d'améliorer la performance du module *Vérificateur* et du module *Reconfigureur*.

CONCLUSION GÉNÉRALE ET PERSPECTIVES

Success is not the key to happiness. Happiness is the key to success. If you love what you are doing, you will be successful.

Albert SCHWEITZER

SYNTHÈSE DES TRAVAUX

Les approches par composants sont un paradigme de programmation bien adapté pour concevoir des applications complexes. En particulier, en raison de la séparation des préoccupations qu'elles favorisent, elles permettent la construction d'applications basées sur des codes très hétérogènes. Avec ce paradigme, le développement d'une application consiste à assembler de nombreux différents composants et représente une tâche difficile pour les utilisateurs et spécialement les non informaticiens. Ainsi, il est important de concevoir des outils efficaces pour aider l'utilisateur à concevoir son application et à vérifier certaines propriétés comme la vivacité.

Dans le cadre de cette thèse, nous nous sommes intéressés à la construction, la vérification et la reconfiguration des applications de visualisation scientifique interactives. Étant donnés des composants hétérogènes, l'assemblage et la composition de ces derniers, peut créer des blocages durant l'exécution de l'application construite. De ce fait, il est nécessaire de se baser sur des mécanismes formels automatiques pour vérifier la composition de ces composants et garantir une application sans blocage. C'est dans cette optique, que nous avons mené l'étude rapportée dans cette thèse qui s'est donc attaquée à la mise en place d'une approche formelle de modélisation, d'analyse, de vérification et de reconfiguration des applications de visualisation scientifique interactives à base de composants. Cette approche formelle augmente le degré de raisonnement rigoureux de certaines propriétés structurelles de nos applications.

Les contributions de cette thèse peuvent se présenter selon les points suivants :

1. La proposition d'une approche formelle des applications de visualisation scientifique interactives ;

2. La proposition d'une méthode pour étudier et garantir la vivacité des applications à base de composants étudiées ;
3. La proposition de la reconfiguration à la volée des applications de visualisation scientifique interactives.

Nous avons proposé, dans un premier temps, un modèle par composants itératifs ComSA (Component-based approach for Scientific Applications) basé sur un système de coordination exogène permettant de spécifier des applications de visualisation scientifique interactives. Nous avons aussi défini une classe particulière des réseaux de Petri appelée sCFN (strict Colored FIFO Nets). Nous avons donné un algorithme permettant de décrire la sémantique opérationnelle d'une application définie dans notre modèle par un sCFN. Cette modélisation représente une étape importante qui nous permet d'obtenir une description formelle des applications de visualisation scientifique interactives qui repose sur des outils solides dans le domaine de la vérification [9, 208].

La modélisation proposée permet de modéliser les différents éléments du modèle ComSA comme les composants avec leur comportement dynamique fourni grâce aux relations d'incidence et les connecteurs qui assemblent les composants pour obtenir une application. Notre modélisation permet de décrire le comportement dynamique de nos applications à l'aide de la sémantique formelle apportée par les sCFN. Ces derniers représentent la base pour analyser et vérifier la composition et l'exécution des applications ComSA et pour corriger leur reconfiguration.

En utilisant les sCFN, nous avons abordé le problème de la vivacité des applications ComSA. En général, les applications à base de composants peuvent rencontrer des blocages lors de leur construction, de leur exécution ou de leur démarrage. Pour cela, nous avons proposé une méthode basée sur les sCFN pour étudier la vivacité des applications ComSA dans le but de garantir qu'elles peuvent démarrer correctement sans contenir de blocages. La détection de blocages que nous avons proposée est basée sur une condition suffisante pour qu'un sCFN soit vivant. Les étapes suivies pour mettre en œuvre cette étude de vivacité sont : (1) Vérifier si les composants satisfont la condition ACN en se basant sur les graphes d'applications, (2) transformer le sCFN obtenu vers un problème SAT afin de détecter les blocages et (3) définir la condition de démarrage pour permettre aux applications de se lancer sans contenir des blocages pendant l'exécution [10, 11, 209].

Parallèlement à l'étude de la vivacité des applications ComSA, nous nous sommes également intéressés au problème de la reconfiguration. L'objectif est de reconfigurer dynamiquement une application sans l'arrêter complètement. Le challenge est de pouvoir identifier la partie influencée par l'insertion, la suppression ou le remplacement d'un composant pour assurer la disponibilité des services non impactés par la reconfiguration. Pour ce faire, nous avons proposé une approche pour appliquer une reconfiguration à la volée. Cette approche permet de minimiser le nombre des services indisponibles d'une application en cours d'exécution en exploitant les communications dirigées par les données et la coordination exogène que le modèle ComSA favorisent. Une telle reconfiguration repose sur les étapes suivantes : (1) la création de la requête

de l'utilisateur pour une reconfiguration souhaitable, (2) la vérification que la nouvelle architecture de l'application à reconfigurer ne va pas contenir de blocages, (3) la délimitation de la région de sécurité, c-à-d la partie impactée par l'opération de la reconfiguration, afin de la mettre en pause et (4) l'application de la reconfiguration et le démarrage de la nouvelle partie reconfigurée [12].

Pour valider nos travaux, nous avons présenté une plateforme appelée ComSATool (Tool Component-based approach for Scientific Applications). Cette dernière est destinée aux applications issues d'*ExaviZ* et permet de modéliser les applications ComSA à l'aide des graphes d'application et les transformer vers les sCFN. De plus, ComSA permet de vérifier la vivacité des applications construites et définir une configuration de démarrage pour les démarrer avec la possibilité de réaliser une simulation des jetons en temps réel des sCFN. Notre outil permet également d'effectuer une reconfiguration dynamique correcte des applications ComSA.

Nos travaux ont conduit à 1 publication dans une revue internationale [209], 1 publication dans une revue nationale [208], 3 publications dans des conférences internationales [10–12] et 1 publications dans une conférence nationale [9]. Nous avons aussi développé un logiciel regroupant ces différentes contributions associées à l'approche ComSA avec des outils de modélisation, de vérification et de reconfiguration.

PERSPECTIVES

Les travaux de recherches présentés dans ce manuscrit ont pu donner des réponses à nos questions que nous nous sommes posées, mais ont également tracé plusieurs perspectives pour poursuivre nos travaux.

Pour enrichir le modèle ComSA, nous envisageons d'introduire la notion du composant composite pour faciliter la description des applications ComSA et de remplacer un ensemble de composants avec un seul composant hiérarchique. Cet hiérarchie décrit qu'un ensemble de composants doit être traité de la même manière comme un seul composant. L'intention d'un composant composite est d'assembler des composants en graphe d'application réduit afin de faciliter la vérification de plusieurs propriétés telles que la vivacité. Ces résultats des composants composites sont génériques, composables et réutilisables. En effet, la mise en œuvre des composants composites permet aux utilisateurs de traiter des composants individuels et des compositions uniformément dans des applications optimales au niveau de nombre de composants.

Un autre défi très important serait de pouvoir étudier la cohérence de données pour améliorer la performance puisque le modèle ComSA dispose des schémas de communication permettant la perte de données. Cette analyse pourrait compter sur le travail [140] qui définit des moyens pour contrôler la perte de données en se basant sur une méthode définissant automatiquement des connecteurs spéciaux nommés régulateurs pour remplir les contraintes de cohérence de données. Ces régulateurs peuvent se modéliser avec les réseaux colorés FIFO stricts afin d'analyser cette cohérence en se basant sur : (1) le numéro d'itération affecté à chaque donnée à l'aide de la place P_{it}

de chaque composant et (2) la désynchronisation de ce numéro à cause de la perte de données.

Par ailleurs, concernant les réseaux FIFO colorés stricts, il serait intéressant de vérifier des propriétés temporelles sur les applications ComSA et particulièrement au niveau des transitions des sCFN. En effet, nous pourrions étendre les sCFN en attachant une contrainte temporelle de type intervalle à chaque transition ce qui pourrait introduire les réseaux FIFO colorés stricts temporels (Temporal strict Colored FIFO Nets, TsCFN). Dans les TsCFN, les transitions contiennent une contrainte de temps qui pourrait représenter, par exemple, la fréquence des composants. Cette contrainte a été implémentée dans ComSATool au niveau de l'animation des sCFN dont l'utilisateur peut définir une durée t représentée en secondes, c-à-d la transition peut être activée après t secondes s'elle est activable. En effet, on peut étendre la notion de vivacité en prenant en considération explicitement cette contrainte temporelle.

D'autre part, il serait intéressant de pouvoir aborder la réduction des sCFN en présentant des règles permettant d'obtenir des sCFN réduits. Ces sCFN réduits devront être équivalents au sCFN de départ pour simplifier l'analyse des applications ComSA. La réduction des sCFN doit permettre de réaliser d'une manière facile les calculs afin de détecter les siphons et les siphons contenant des trappes.

PUBLICATIONS CONNEXES

A. Ait Wakrime, S. Limet et S. Robert, *On the Fly Reconfiguration of Interactive Scientific Visualization Applications*, International Conference on High Performance Computing & Simulation (HPCS), 20-24 July 2015, Amsterdam, the Netherlands.

A. Ait Wakrime, S. Limet et S. Robert, *Modeling Interactive Scientific Visualization Applications with Strict Colored FIFO Nets*, Global Journal of Engineering Science and Researches (GJESR), vol.2, 2015.

A. Ait Wakrime, S. Limet et S. Robert, *Modeling Interactive Scientific Visualization Applications with Strict Colored FIFO Nets*, the 2nd International Workshop on Software Engineering and Systems Architecture (SESA), 13 December 2014, Tetouan, Morocco.

A. Ait Wakrime, S. Limet et S. Robert, *Place-Liveness of ComSA Applications*, the 11th International Symposium on Formal Aspects of Component Software (FACS), LNCS 8997, 10-12 September 2014, Bertinoro, Italy.

A. Ait Wakrime, *Deadlock-freedom of component-based approach for scientific applications*, Summer School on Modelling and Verification of Parallel Processes (MOVEP), 7-11 July 2014, Nantes, France.

A. Ait Wakrime, S. Limet et S. Robert, *Réseaux FIFO Colorés Stricts pour la formalisation des applications de visualisation scientifique interactives*, Revue des Nouvelles Technologies de l'Information (RNTI), vol. RNTI-L-7, 2014.

A. Ait Wakrime, S. Limet et S. Robert, *Modélisation des applications de visualisation scientifique interactives*, Conférence francophone sur les Architectures Logicielles (CAL), 30-31 May 2013, Toulouse, France.

Annexes

REPRÉSENTATION DE LA DESCRIPTION DE L'ARCHITECTURE ET DE COMPORTEMENTS DES APPLICATIONS ComSA.



L'entrée principale du ComSATool est un graphe d'application. Nous avons utilisé le langage de description XML pour la représentation de la description de l'architecture et de comportements. Un graphe d'application est un fichier XML qui a comme racine l'élément `<application>`. Cet élément regroupe un ensemble d'éléments `<component>`, `<connector>` et `<link>`.

L'élément `<component>` représente un composant de l'application et est défini par les éléments suivants :

- `<inputports>` représente les ports d'entrée. Cet élément spécifie un ensemble d'élément `<port>`.
 - L'élément `<port>` définit le nom du port d'entrée.
- `<outputports>` représente les ports de sortie. Cet élément contient un ensemble d'élément `<port>`
 - L'élément `<port>` définit le nom du port de sortie.
- `<relationship>` spécifie l'ensemble des relations d'incidence du composant. Cet élément contient deux éléments `<RIin>` et `<RIout>`.
 - L'élément `<RIin>` définit les ports d'entrée de la relation d'incidence et contient un ensemble d'éléments `<port>` appartenant à l'élément `<inputports>`.
 - L'élément `<RIout>` définit les ports de sortie de la relation d'incidence et contient un ensemble d'éléments `<port>` appartenant à l'élément `<inputports>`.

Au niveau de l'élément `<component>`, il existe des attributs associés aux plusieurs éléments à savoir :

- L'attribut `name` est associé à l'élément `<component>` pour définir son nom.
- L'attribut `id` est affecté à l'élément `<relationship>` dont la valeur est un identifiant unique.
- L'attribut `ispaused` appartient à l'élément `<relationship>` dont la valeur est un booléen pour indiquer si la relation d'incidence est en pause.

L'élément `<connector>` spécifie un connecteur et est défini par les éléments suivants :

- `<inputports>` représente les ports d'entrée. Cet élément spécifie un ensemble d'élément `<port>`.
 - L'élément `<port>` définit le nom du port d'entrée.
- `<outputports>` représente les ports de sortie. Cet élément contient un ensemble d'élément `<port>`.
 - L'élément `<port>` définit le nom du port de sortie.

Au niveau de l'élément `<connector>`, des attributs sont associés à ce dernier à savoir :

- L'attribut `name` pour définir le nom du connecteur.
- L'attribut `type` symbolise le type du connecteur dont la valeur peut être soit `sfifo`, soit `bbuffer`, soit `nbuffer`, soit `greedy` ou soit `nbgreedy`.

L'élément `<link>` spécifie un lien connectant un connecteur avec un composant et est défini par les éléments suivants :

- `<src>` identifie le point de départ du lien. Cet élément contient deux éléments `<objet>` et `<port>`.
 - L'élément `<objet>` spécifie le nom de l'objet source qui peut être un composant ou un connecteur.
 - L'élément `<port>` identifie le nom du port utilisé de l'élément `<objet>` source.
- `<dest>` identifie le point de destination du lien. Cet élément contient deux éléments `<objet>` et `<port>`.
 - L'élément `<objet>` spécifie le nom de l'objet destination qui peut être un composant ou un connecteur.
 - L'élément `<port>` identifie le nom du port utilisé de l'élément `<objet>` destination.

Pour l'élément `<link>` deux attributs sont utilisés :

- L'attribut `id` mentionne l'identifiant unique du lien.
- L'attribut `type` indique le type du lien qui peut être un lien de données ou un lien de déclenchement. Dans le premier cas l'attribut prend la valeur `DL` et dans le deuxième cas l'attribut prend la valeur `TL`.

Le listing A.1 montre un extrait d'un graphe d'application contenant deux composant I_1 et S_1 connectés à l'aide d'un connecteur nbG_2 du type `nbGreedy`.

```

<application>
  <component name="I1">
    <inputports>
      <port>s</port>
      <port>f</port>
    </inputports>
    <outputports>
      <port>e</port>
      <port>P3d</port>
      <port>c1</port>
    </outputports>
    <relationship id="r1" ispaused="false">
      <Rlout>
        <port>P3d</port>
      </Rlout>
    </relationship>
    <relationship id="r2" ispaused="false">
      <Rlout>
        <port>c1</port>
      </Rlout>
    </relationship>
    <relationship id="r3" ispaused="false">
      <Rlin>
        <port>f</port>
      </Rlin>
    </relationship>
  </component>
  <component name="S1">
    <inputports>
      <port>s</port>
      <port>P3d</port>
      <port>c1</port>
      <port>c2</port>
      <port>an</port>
    </inputports>
    <outputports>
      <port>e</port>
      <port>av</port>
      <port>act1</port>
      <port>act2</port>
    </outputports>
    <relationship id="r1" ispaused="false">
      <Rlin>
        <port>P3d</port>
      </Rlin>
      <Rlout>
        <port>av</port>
      </Rlout>
    </relationship>
  </component>
</application>

```

```

</relationship>
<relationship id="r2" ispaused="false">
  <Rlin>
    <port>c1</port>
  </Rlin>
  <Rlout>
    <port>act2</port>
  </Rlout>
</relationship>
<relationship id="r3" ispaused="false">
  <Rlin>
    <port>c2</port>
    <port>an</port>
  </Rlin>
  <Rlout>
    <port>act1</port>
  </Rlout>
</relationship>
</component>
<connector name="nbG2" type="nbgreedy">
  <inputports>
    <port>i</port>
    <port>s</port>
  </inputports>
  <outputports>
    <port>o</port>
  </outputports>
</connector>
<link id="L47436" type="DL">
  <src>
    <objet>I1</objet>
    <port>P3d</port>
  </src>
  <dest>
    <objet>nbG2</objet>
    <port>i</port>
  </dest>
</link>
<link id="L25366" type="TL">
  <src>
    <objet>s1</objet>
    <port>e</port>
  </src>
  <dest>
    <objet>nbG2</objet>
    <port>s</port>
  </dest>
</link>
<link id="L76238" type="DL">

```



```
<src>
  <objet>nbG2</objet>
  <port>o</port>
</src>
<dest>
  <objet>s1</objet>
  <port>P3d</port>
</dest>
</link>
</application>
```

Listing A.1 – Extrait de l'architecture de l'application de la figure 6.15.

EXEMPLE D'UN FICHIER .DIMACS

```

p cnf 11235 54436
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92
 93 94 95 96 97 98 99 100 101 102 103 104 105 0
-1 3 0
-1 3 0
-1 6 3 0
-2 1 0
-3 2 0
-4 3 0
-4 6 3 0
-5 3 0
-5 6 3 0
-6 66 0
-7 9 0
-7 9 0
-8 7 0
-9 8 0
-10 9 0
-11 9 0
-12 15 14 0
-12 16 14 0
-12 15 16 14 0
-12 17 18 14 0
-12 15 17 18 14 0
-12 16 17 18 14 0
-12 15 16 17 18 14 0
-13 12 0
-14 13 0
-15 50 0
-16 54 0

```

```
-17 58 0
-18 62 0
-19 15 14 0
-19 15 16 14 0
-19 15 17 18 14 0
-19 15 16 17 18 14 0
-20 16 14 0
-20 15 16 14 0
-20 16 17 18 14 0
-20 15 16 17 18 14 0
-21 17 18 14 0
-21 15 17 18 14 0
-21 16 17 18 14 0
-21 15 16 17 18 14 0
-22 24 0
-22 26 24 0
-23 22 0
-24 23 104 0
-25 24 0
-25 26 24 0
-26 103 0
-27 26 24 0
-28 31 32 33 30 0
-28 31 33 34 30 0
-28 31 32 33 34 30 0
-29 28 0
-30 29 105 0
-31 90 0
-32 86 0
-33 94 0
-34 97 0
-35 31 32 33 30 0
-35 31 33 34 30 0
-35 31 32 33 34 30 0
-36 39 38 0
-37 36 0
-38 37 0
-39 100 0
```

Listing B.1 – Extrait du fichier .dimacs de l'application de la figure 6.15.

BIBLIOGRAPHIE

- [1] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.
Cité page 58.
- [2] ISaGRAF, ICS Triplex ISaGRAF Inc. <http://www.isagraf.com/>, 2010.
Cité page 31.
- [3] IEC 61131-3 Programmable controllers - Part 3 : Programming languages. <https://webstore.iec.ch/publication/4552>, 2013.
Cité page 31.
- [4] 4DIAC-RTE (FORTE) : IEC 61499 Compliant Runtime Environment, PROFAC-TOR Produktionsforschungs GmbH. <http://www.fordiac.org/>, 2014.
Cité page 31.
- [5] Function Block Development Kit, Holobloc Inc. <http://www.holobloc.com/>, 2015.
Cité page 31.
- [6] *CORBA Component Model Specification*. OMG Available Specification Version 4.0, April 2006.
Cité page 25.
- [7] M. Abadi and A. Gordon. A calculus for cryptographic protocols : The spi calculus, digital. *Systems Research Center*, (149), 1998.
Cité page 62.
- [8] F. Abouzaid. Toward a pi-calculus based verification tool for web services orchestrations. In *Computer Supported Activity Coordination*, pages 23–34, 2006.
Cité page 62.
- [9] A. Ait Wakrime, S. Limet, and S. Robert. Modélisation des applications de visualisation scientifique interactives. In *CAL 2013*, page 1, Toulouse, France, May 2013.
Cité pages 154 et 155.
- [10] A. Ait Wakrime, S. Limet, and S. Robert. Modeling Interactive Scientific Visualization Applications with Strict Colored FIFO Nets. In *SESA 2014*, Proceedings of the 2014 International Workshop on Software Engineering and Systems Architecture, Tetouan, Morocco, Dec 2014.
Cité pages 154 et 155.

- [11] A. Ait Wakrime, S. Limet, and S. Robert. Place-liveness of ComSA Applications. In *FACS, LNCS*, pages 1–18, Bertinoro, Italy, Sep 2014. Springer.
Cité pages 154 et 155.
- [12] A. Ait Wakrime, S. Limet, and S. Robert. On the Fly Reconfiguration of Interactive Scientific Visualization Applications. In IEEE, editor, *HPCS 2015, Proceedings of the 2015 International Conference on High Performance Computing & Simulation*, Amsterdam, Netherlands, Jul 2015.
Cité page 155.
- [13] R. Akkiraju, J. Farrell, J. A. Miller, M. Nagarajan, A. Sheth, K. Verma, R. Akkiraju, J. Farrell, J. A. M. Iller, M. Nagarajan, A. Sheth, and K. Verma. Web service semantics – wsdl-s. In *In W3C Workshop on Frameworks for Semantic in Web Services*, 2005.
Cité page 41.
- [14] D. L. Aksnes, K. B. Ulvestad, B. M. Baliño, J. Berntsen, J. K. Egge, and E. Svendsen. Ecological modelling in coastal waters : towards predictive physical-chemical-biological simulation models. *Ophelia*, 41(1) :5–36, 1995.
Cité page 1.
- [15] J. Aldrich, C. Chambers, and D. Notkin. Archjava : Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering, ICSE '02*, pages 187–197, New York, NY, USA, 2002. ACM.
Cité page 25.
- [16] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. Flowvr : a middleware for large scale virtual reality applications. In *Proceedings of Euro-par 2004*, Pisa, Italia, August 2004.
Cité pages 47 et 89.
- [17] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.
Cité page 25.
- [18] S. AlSairafi, F. Emmanouil, M. Ghanem, N. Giannadakis, Y. Guo, D. Kalaitzopoulos, M. Osmond, A. Rowe, J. Syed, and P. Wendel. The design of discovery net : Towards open grid services for knowledge discovery. *IJHPCA*, 17(3) :297–315, 2003.
Cité page 37.
- [19] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler : an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424, June 2004.
Cité page 43.
- [20] P. Alvarez, J. A. Banares, and J. Ezpeleta. Approaching web service coordination and composition by means of petri nets. the case of the nets-within-nets para-

- digm. In *Service-Oriented Computing-ICSOC 2005*, pages 185–197. Springer, 2005.
Cité page 63.
- [21] G. A. Araújo, F. H. Carvalho Jr, and R. C. Corrêa. Implementing endogenous and exogenous connectors with the common component architecture. In *Proceedings of the 2009 Workshop on Component-Based High Performance Computing*, page 12. ACM, 2009.
Cité page 24.
- [22] F. Arbab. Reo : A channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3) :329–366, June 2004.
Cité page 79.
- [23] A. Arbeláez, A. Aristizábal, J. Gutiérrez, H. A. López, J. A. Pérez, C. Rueda, and F. D. Valencia. Process calculi to analyze emerging applications in concurrency. *Ensenanza*, 8(1), 2000.
Cité page 61.
- [24] H. Aref, R. D. Charles, and T. T. Elvins. Frontiers of scientific visualization. chapter Scientific Visualization of Fluid Flow, pages 7–44. John Wiley & Sons, Inc., New York, NY, USA, 1994.
Cité page 13.
- [25] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 115–124. IEEE, 1999.
Cité page 33.
- [26] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren. The cca component model for high-performance scientific computing. *Concurr. Comput. : Pract. Exper.*, 18(2) :215–229, Feb. 2006.
Cité pages 3 et 33.
- [27] P. C. Attie, S. Bensalem, M. Bozga, M. Jaber, J. Sifakis, and F. A. Zaraket. An abstract framework for deadlock prevention in bip. In *Formal Techniques for Distributed Systems*, pages 161–177. Springer, 2013.
Cité page 89.
- [28] T. Aubonnet, L. Henrio, S. Kessal, O. Kulankhina, F. Lemoine, E. Madelaine, C. Ruz, and N. Simoni. Management of service compositionbased on self-controlled components. *Journal of Internet Services and Applications*, 6(1) :1–17, 2015.
Cité page 59.
- [29] S. Aukstaklnis and D. Blatner. *Silicon Mirage; The Art and Science of Virtual Reality*. Peachpit Press, Berkeley, CA, USA, 1992.
Cité page 14.

- [30] R. Axelrod. Advancing the art of simulation in the social sciences. In *Simulating social phenomena*, pages 21–40. Springer, 1997.
Cité page 1.
- [31] R. Balasubramanian, S. Babak, D. Churches, and T. Cokelaer. Geo 600 online detector characterization system. *Classical and Quantum Gravity*, 22 :4973–4985, dec 2005.
Cité page 40.
- [32] J. Banks, J. Carson, and B. Nelson. *DM Nicol, Discrete-Event System Simulation*. Prentice hall Englewood Cliffs, NJ, USA, 2000.
Cité page 57.
- [33] T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model-checking distributed components : The vercors platform. *Electronic Notes in Theoretical Computer Science*, 182 :3–16, 2007.
Cité page 62.
- [34] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In *Model Checking Software*, pages 154–168. Springer, 2005.
Cité page 28.
- [35] R. Bastide and E. Barboni. Software components : a formal semantics based on coloured petri nets. *Electronic Notes in Theoretical Computer Science*, 160 :57–73, 2006.
Cité pages 63 et 89.
- [36] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, SEFM '06*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
Cité page 31.
- [37] A. Basu, L. Mounier, M. Poulhies, J. Pulou, and J. Sifakis. Using bip for modeling and verification of networked systems—a case study on tinyos-based networks. In *Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on*, pages 257–260. IEEE, 2007.
Cité page 31.
- [38] F. Baude. A perspective on the coregrid grid component model. In M. Alexander, P. D’Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Di Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. Scott, J. Traff, G. Vallée, and J. Weidendorfer, editors, *Euro-Par 2011 : Parallel Processing Workshops*, volume 7155 of *Lecture Notes in Computer Science*, pages 115–116. Springer Berlin Heidelberg, 2012.
Cité page 46.
- [39] P. Behm, P. Desforgues, and J.-M. Meynadier. Météor : An industrial success in formal development. In *B’98 : Recent Advances in the Development and Use of the B Method*, pages 26–26. Springer, 1998.
Cité page 57.

- [40] M.-L. Benalycherif and C. Girault. Behavioural and structural composition rules preserving liveness by synchronization for colored fifo nets. In *Application and Theory of Petri Nets*, pages 73–92, 1996.
Cité page 92.
- [41] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. Compositional verification for component-based systems and application. *IET software*, 4(3) :181–193, 2010.
Cité page 31.
- [42] S. Bensalem, A. Griesmayer, A. Legay, T.-H. Nguyen, and D. Peled. Efficient deadlock detection for concurrent systems. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 119–129, July 2011.
Cité page 4.
- [43] K. Bergner, A. Rausch, and M. Sihling. Componentware - the big picture. In *Proceedings of the International Workshop on Component-Based Software Engineering, CBSE 98*, Apr 1998.
Cité page 23.
- [44] J. Bigot, Z. Hou, C. Pérez, and V. Pichon. A low level component model easing performance portability of HPC applications. *Computing*, 96(12) :1115–1130, Dec 2014.
Cité page 32.
- [45] G. Black and V. Vyatkin. Intelligent component-based automation of baggage handling systems with iec 61499. *Automation Science and Engineering, IEEE Transactions on*, 7(2) :337–351, 2010.
Cité page 31.
- [46] B. Blanchet. Automatic verification of security protocols in the symbolic model : The verifier proverif. In *Foundations of Security Analysis and Design VII*, pages 54–87. Springer, 2014.
Cité page 61.
- [47] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN systems*, 14(1) :25–59, 1987.
Cité page 62.
- [48] G. Booch, I. Jacobson, and J. Rumbaugh. Omg unified modeling language specification. *Object Management Group*, 1034 :15–44, 2000.
Cité page 23.
- [49] M. Boreale and D. Gorla. Process calculi and the verification of security protocols. *Journal of Telecommunications and Information Technology*, pages 28–39, 2002.
Cité page 61.
- [50] H. Bouziane, C. Pérez, and T. Priol. A software component model with spatial and temporal compositions for grid infrastructures. In E. Luque, T. Margalef, and D. Benítez, editors, *Euro-Par 2008 – Parallel Processing*, volume 5168 of *Lecture*

- Notes in Computer Science*, pages 698–708. Springer Berlin Heidelberg, 2008.
Cité pages 3 et 46.
- [51] H. L. Bouziane, C. Pérez, and T. Priol. A software component model with spatial and temporal compositions for grid infrastructures. In *Euro-Par 2008–Parallel Processing*, pages 698–708. Springer, 2008.
Cité page 51.
- [52] D. A. Bowman. *Interaction Techniques for Common Tasks in Immersive Virtual Environments : Design, Evaluation, and Application*. PhD thesis, Atlanta, GA, USA, 1999.
Cité page 15.
- [53] M. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol verification with the aldebaran toolset. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1) :166–183, 1997.
Cité page 31.
- [54] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The if toolset. In *Formal Methods for the Design of Real-Time Systems*, pages 237–267. Springer, 2004.
Cité page 31.
- [55] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 51–59. IEEE, 2000.
Cité page 33.
- [56] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1) :115–131, 1988.
Cité page 59.
- [57] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12) :1257–1284, Sept. 2006.
Cité page 27.
- [58] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy : A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, April 1994.
Cité page 43.
- [59] J.-M. Burkhardt. Immersion, réalisme et présence dans la conception et l'évaluation des environnements virtuels = immersion, realism and presence in the design and evaluation of virtual environments. *Psychologie française*, 48(2) :35–42, Nov. 2003.
Cité page 15.
- [60] S. K. Card, J. D. Mackinlay, and B. Shneiderman. *Readings in information visualization : using vision to think*. Morgan Kaufmann, 1999.
Cité page 16.

- [61] Y. Cardinale, J. El Haddad, M. Manouvrier, and M. Rukoz. Web service composition based on petri nets : Review and contribution. In *Resource Discovery*, pages 83–122. Springer, 2013.
Cité page 63.
- [62] D. Casper. The nuance neutrino physics simulation, and the future. *Nuclear Physics B-Proceedings Supplements*, 112(1) :161–170, 2002.
Cité page 1.
- [63] B. Charroux, A. Osmani, and Y. Thierry-Mieg. *UML 2 : Pratique de la Modélisation*. Synthex Informatique. Pearson, 3 eme (288 pages) edition, 2010.
Cité page 55.
- [64] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vincentelli, and J. Rabae. Embedded system design using uml and platforms. In *System Specification & Design Languages*, pages 119–128. Springer, 2003.
Cité page 56.
- [65] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
Cité page 59.
- [66] W. W. W. Consortium et al. Extensible markup language (xml) 1.0. W3C XML, February, 1998.
Cité page 21.
- [67] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.
Cité page 61.
- [68] M. Corporation. Component Object Model Specification (Version 0.9). Technical report, Microsoft, October 1995.
Cité page 22.
- [69] T. Coupaye and J.-B. Stefani. Fractal component-based software engineering. In *Proceedings of the 2006 Conference on Object-oriented Technology : ECOOP 2006 Workshop Reader, ECOOP’06*, pages 117–129, Berlin, Heidelberg, 2007. Springer-Verlag.
Cité page 27.
- [70] V. Curcin and M. Ghanem. Scientific workflow systems - can one size fit all ? In *Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International*, pages 1–9, Dec 2008.
Cité pages 37 et 39.
- [71] L. D. Da Silva, A. Perkusich, and K. Gorgonio. *Petri Nets for Component-Based Software Systems Development*. INTECH Open Access Publisher, 2008.
Cité page 63.
- [72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., London, UK, UK, 1972.
Cité page 20.

- [73] P.-C. David and T. Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In *Software Composition*, pages 82–97. Springer, 2006.
Cité page 50.
- [74] J. Davies. *Specification and Proof in Real Time CSP*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.
Cité page 61.
- [75] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, July 1962.
Cité page 61.
- [76] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3) :201–215, 1960.
Cité page 61.
- [77] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus : A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3) :219–237, July 2005.
Cité page 42.
- [78] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, and K. Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, (o) :–, 2014.
Cité pages 42 et 43.
- [79] X. Deng, Z. Lin, W. Cheng, R. Xiao, L. Fang, and L. Li. Modeling web service choreography and orchestration with colored petri nets. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 2, pages 838–843. IEEE, 2007.
Cité page 63.
- [80] J. Dent and P. Thornton. The role of biological simulation models in farming systems research. *Agricultural Administration and Extension*, 29(2) :111–122, 1988.
Cité page 1.
- [81] M. Diaz. *Petri Nets : Fundamental Models, Verification and Applications*. Wiley-IEEE Press, 2009.
Cité pages x et 64.
- [82] K. Dokter, S.-S. Jongmans, F. Arbab, and S. Bliudze. Relating bip and reo. *arXiv preprint arXiv :1508.04848*, 2015.
Cité page 24.
- [83] J. Dormoy, O. Kouchnarenko, and A. Lanoix. Using temporal logic for dynamic reconfigurations of components. In *Formal Aspects of Component Software*, pages

- 200–217. Springer, 2012.
Cité page 49.
- [84] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML : The Catalysis Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
Cité page 22.
- [85] W. W. Eckerson. Three tier client/server architecture : Achieving scalability, performance, and efficiency in client server applications. *Open Information Systems*, 10(1), 1995.
Cité page 20.
- [86] S. R. Ellis. What are virtual environments ? *IEEE Comput. Graph. Appl.*, 14(1) :17–22, Jan. 1994.
Cité pages 14 et 15.
- [87] T. Epperly, S. R. Kohn, and G. Kumfert. Component technology for high-performance scientific simulation software. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, pages 69–86, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.
Cité page 33.
- [88] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with agwl : an abstract grid workflow language. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 676–685. IEEE, 2005.
Cité page 46.
- [89] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL) : An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.
Cité page 25.
- [90] A. Finkel and G. Memmi. FIFO nets : A new model of parallel computation. In *Theoretical Computer Science*, volume 145 of LNCS, pages 111–121. Springer, 1982.
Cité page 92.
- [91] P. Fuchs, B. Arnaldi, and J. Tisseau. *La réalité virtuelle et ses applications*. Sep 2003.
Cité pages 14 et 15.
- [92] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. Icen : Optimisation of component applications within a grid environment. *Parallel Computing*, 28(12) :1753–1772, 2002.
Cité page 45.
- [93] J. L. Gabbard, D. Hix, and J. E. Swan. User-centered design and evaluation of virtual environments. *IEEE Comput. Graph. Appl.*, 19(6) :51–59, Nov. 1999.
Cité page 15.
- [94] D. Garlan, R. Monroe, and D. Wile. Acme : An architecture description interchange language. In *Proceedings of the 1997 Conference of the Centre for Advanced*

- Studies on Collaborative Research, CASCON '97*, pages 7–. IBM Press, 1997.
Cité page 25.
- [95] N. Gaspar, L. Henrio, and E. Madelaine. Formally reasoning on a reconfigurable component-based system—a case study for the industrial world. In *Formal Aspects of Component Software*, pages 137–156. Springer, 2014.
Cité page 49.
- [96] K. Geebelen, S. Michiels, and W. Joosen. Dynamic reconfiguration using template based web service composition. In *Proceedings of the 3rd workshop on Middleware for service oriented computing*, pages 49–54. ACM, 2008.
Cité page 48.
- [97] N. Gershon. From perception to visualization. *Scientific Visualization*, 1994.
Cité page 13.
- [98] M. M. Ghanem, Y. Guo, H. Lodhi, and Y. Zhang. Automatic scientific text classification using local patterns : Kdd cup 2002 (task 1). *SIGKDD Explor. Newsl.*, 4(2) :95–96, Dec. 2002.
Cité page 37.
- [99] J. J. Gibson. The ecological approach to the visual perception of pictures. *Leonardo*, pages 227–235, 1978.
Cité page 16.
- [100] A. Girard and G. J. Pappas. Verification using simulation. In *Hybrid Systems : Computation and Control*, pages 272–286. Springer, 2006.
Cité page 57.
- [101] P. Godefroid, M. Y. Levin, and D. Molnar. Sage : whitebox fuzzing for security testing. *Queue*, 10(1) :20, 2012.
Cité page 58.
- [102] J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team. Galaxy : a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8) :R86, 2010.
Cité page 35.
- [103] A. Gokhale and D. C. Schmidt. Principles for optimizing corba internet inter-orb protocol performance. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 7, pages 376–385. IEEE, 1998.
Cité page 25.
- [104] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The cactus framework and toolkit : Design and applications. In *Proceedings of the 5th International Conference on High Performance Computing for Computational Science, VECPAR'02*, pages 197–227, Berlin, Heidelberg, 2003. Springer-Verlag.
Cité page 3.
- [105] G. Gössler and J. Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3) :161–183, Mar. 2005.
Cité page 3.

- [106] R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik. Model-driven web services development. In *e-Technology, e-Commerce and e-Service, 2004. EEE'04. 2004 IEEE International Conference on*, pages 42–45. IEEE, 2004.
Cité page 56.
- [107] G. Guiho and C. Hennebert. Sacem software validation. In *Proceedings of the 12th International Conference on Software Engineering, ICSE '90*, pages 186–191, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.
Cité page 57.
- [108] Y. Guo, J. G. Liu, M. Ghanem, K. Mish, V. Curcin, C. Haselwimmer, D. Sotiriou, K. K. Muraleetharan, and L. Taylor. Bridging the macro and micro : A computing intensive earthquake study using discovery net. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC '05*, pages 68–, Washington, DC, USA, 2005. IEEE Computer Society.
Cité page 37.
- [109] D. Hagimont, P. Stolf, L. Broto, and N. De Palma. Component-based autonomic management for legacy software. In *Autonomic Computing and Networking*, pages 83–104. Springer, 2009.
Cité page 28.
- [110] A. Hall. Seven myths of formal methods. *Software, IEEE*, 7(5) :11–19, 1990.
Cité page 59.
- [111] N. Hameurlain. Controllability preservation and behavioural refinement for service protocols. In *Services Computing Conference (APSCC), 2012 IEEE Asia-Pacific*, pages 203–210. IEEE, 2012.
Cité page 89.
- [112] N. Hameurlain. A compositional framework to the specification of service protocols controllability and substitutability. *International Journal of Computer Mathematics*, 91(6) :1137–1155, 2014.
Cité page 63.
- [113] G. T. Heineman and W. T. Councill, editors. *Component-based Software Engineering : Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
Cité page 21.
- [114] C. Heinzemann, C. Priesterjahn, and S. Becker. Towards modeling reconfiguration in hierarchical component architectures. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pages 23–28. ACM, 2012.
Cité page 49.
- [115] S. Herres-Pawlis, A. Hoffmann, L. De La Garza, J. Krüger, S. Gesing, and R. Grunzke. User-friendly metaworkflows in quantum chemistry. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–3, Sept 2013.
Cité page 35.

- [116] C. Hoare. Algebra and models. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI Series*, pages 161–195. Springer Berlin Heidelberg, 1993.
Cité page 58.
- [117] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8) :666–677, Aug. 1978.
Cité page 61.
- [118] D. Hollingsworth. Workflow management coalition - the workflow reference model. Technical report, Workflow Management Coalition, Jan 1995.
Cité page 35.
- [119] J. Hooman, H. Kugler, I. Ober, A. Votintseva, and Y. Yushtein. Supporting uml-based development of embedded systems by formal techniques. *Software & Systems Modeling*, 7(2) :131–155, 2008.
Cité page 56.
- [120] J.-M. Hufflen. Using model-checking techniques for component-based systems with reconfigurations. *arXiv preprint arXiv :1503.04915*, 2015.
Cité page 59.
- [121] ISO. Information processing systems - open systems interconnection - LOTOS : a formal description technique based on the temporal ordering of observational behaviour = systemes de traitement de l'information - interconnexion de systemes ouverts - LOTOS, 1989.
Cité page 62.
- [122] K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
Cité page 92.
- [123] K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3) :213–254, May 2007.
Cité page 63.
- [124] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.
Cité page 44.
- [125] M. Khalgui. Distributed reconfigurations of autonomous iec61499 systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1) :18, 2013.
Cité page 49.
- [126] J. Kramer and J. Magee. The evolving philosophers problem : Dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11) :1293–1306, 1990.
Cité page 49.
- [127] O. Kreylos, A. M. Tesdall, B. Hamann, J. K. Hunter, and K. I. Joy. Interactive visualization and steering of CFD simulations. In *Proceedings of the symposium on Data Visualisation 2002, VISSYM '02*, pages 25–34, Aire-la-Ville, Switzerland,

- Switzerland, 2002. Eurographics Association.
Cité page 17.
- [128] P. Kruchten, H. Obbink, and J. Stafford. The past, present, and future for software architecture. *IEEE Softw.*, 23(2) :22–30, Mar. 2006.
Cité page 20.
- [129] G. Kumpf, J. Leek, and T. Epperly. Babel remote method invocation. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.
Cité page 33.
- [130] S. Lacour, C. Pérez, and T. Priol. Deploying corba components on a computational grid : General principles and early experiments using the globus toolkit. In W. Emmerich and A. Wolf, editors, *Component Deployment*, volume 3083 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin Heidelberg, 2004.
Cité page 26.
- [131] J. Lacouture and P. Aniot. Self-adaptable discovery and composition of services based on the semantic compaa approach. *Innovations and Approaches for Resilient and Adaptive Systems*, page 150, 2012.
Cité page 50.
- [132] J. Latta and D. Oberg. A conceptual virtual reality model. *Computer Graphics and Applications, IEEE*, 14(1) :23–29, Jan 1994.
Cité page 14.
- [133] J. LaViola, Joseph J., Prabhat, A. S. Forsberg, D. H. Laidlaw, and A. v. Dam. Virtual reality-based interactive scientific visualization environments. In R. Liere, T. Adriaansen, and E. Zudilova-Seinstra, editors, *Trends in Interactive Visualization*, Advanced Information and Knowledge Processing, pages 225–250. Springer London, 2009.
Cité page 17.
- [134] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9) :1235–1245, Sept 1987.
Cité page 44.
- [135] E. A. Lee and S. Neuendorffer. Moml - a modeling markup language in xml - version 0.4, 2000.
Cité page 43.
- [136] C. Li, L. Huang, L. Chen, and W. Luo. Deadlock detection and recovery for component-based systems. *Mathematical and Computer Modelling*, 58(5–6) :1362 – 1378, 2013.
Cité page 4.
- [137] P. Li, J. Castrillo, G. Velarde, I. Wassink, S. Soiland-Reyes, S. Owen, D. Withers, T. Oinn, M. Pocock, C. Goble, S. Oliver, and D. Kell. Performing statistical analyses on quantitative data in taverna workflows : An example using r and maxdbrowse to identify differentially-expressed genes from microarray data. *BMC*

- Bioinformatics*, 9(1), 2008.
Cité page 39.
- [138] S. Limet, S. Robert, and A. Turki. FlowVR-SciViz : a component-based framework for interactive scientific visualization. In *Proceedings of the 2009 Workshop on Component-Based High Performance Computing, CBHPC '09*, pages 17 :1–17 :9, New York, NY, USA, 2009. ACM.
Cité pages 17 et 70.
- [139] S. Limet, S. Robert, and A. Turki. Coherence and performance for interactive scientific visualization applications. In S. Apel and E. K. Jackson, editors, *Software Composition - 10th International Conference, SC 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, volume 6708 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2011.
Cité pages 47 et 70.
- [140] S. Limet, S. Robert, and A. Turki. Controlling an iteration-wise coherence in dataflow. In *Formal Aspects of Component Software*, pages 241–258. Springer, 2012.
Cité pages 5, 70, 76, 80 et 155.
- [141] R. J. Machado and J. M. Fernandes. A petri net meta-model to develop software components for embedded systems. In *Application of Concurrency to System Design, 2001. Proceedings. 2001 International Conference on*, pages 113–122. IEEE, 2001.
Cité page 63.
- [142] J. Magee and J. Kramer. Dynamic structure in software architectures. *ACM SIGSOFT Software Engineering Notes*, 21(6) :3–14, 1996.
Cité page 24.
- [143] C. Mahulea, L. Mahulea, J. García-Soriano, and J. Colom. Petri nets with resources for modeling primary healthcare systems. In *18th International Conference on System Theory, Control and Computing (ICSTCC 2014)*, 17 - 19 October 2014, Sinaia, Romania, 10/2014 2014.
Cité page 63.
- [144] S. Majithia, M. Shields, I. Taylor, and I. Wang. Triana : A graphical web service composition and execution toolkit. In *Proceedings of the IEEE International Conference on Web Services, ICWS '04*, pages 514–, Washington, DC, USA, 2004. IEEE Computer Society.
Cité page 40.
- [145] M. Malawski, J. Meizner, M. Bubak, and P. Gepner. Component approach to computational applications on clouds. *Procedia Computer Science*, 4 :432–441, 2011.
Cité page 30.
- [146] M. Martens and M. Majster-Cederbaum. Deadlock-freedom in component systems with architectural constraints. *Formal Methods in System Design*, 41(2) :129–177, 2012.
Cité page 4.

- [147] S. Matougui and A. Beugnard. Two ways of implementing software connections among distributed components. In *On the Move to Meaningful Internet Systems 2005 : CoopIS, DOA, and ODBASE*, pages 997–1014. Springer, 2005.
Cité page 24.
- [148] B. McCormick, T. Defanti, and M. Brown. Visualization in scientific computing. *Computer Graphics*, 22(6) :103 – 111, 1987.
Cité page 13.
- [149] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1) :70–93, Jan. 2000.
Cité page 24.
- [150] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22Nd International Conference on Software Engineering, ICSE '00*, pages 178–187, New York, NY, USA, 2000. ACM.
Cité page 24.
- [151] K. Michalickova, G. Bader, M. Dumontier, H. Lieu, D. Betel, R. Isserlin, and C. Hogue. Seqhound : biological sequence and structure database as a platform for bioinformatics research. *BMC Bioinformatics*, 3(1), 2002.
Cité page 39.
- [152] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
Cité page 61.
- [153] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1) :1–40, Sept. 1992.
Cité page 62.
- [154] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.
Cité page 61.
- [155] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.
Cité page 58.
- [156] P. Naur and B. Randell, editors. *Software Engineering : Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.
Cité page 20.
- [157] O. Nierstrasz. Research topics in software composition. In A. Napoli, editor, *Actes des journées Langages et Modèles à Objets, LMO'95. Nancy, 12-13 octobre*, pages 193–206. INRIA, 1995.
Cité page 21.
- [158] O. Oanea, H. Wimmel, and K. Wolf. New algorithms for deciding the siphon-trap property. In *Proceedings of the 31st Int. Conf. on Applications and Theory of Petri*

- Nets*, LNCS, pages 267–286. Springer, 2010.
Cité page 120.
- [159] OASIS. Web Services Business Process Execution Language Version 2.0. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html/>, 2007.
Cité page 28.
- [160] OASIS. Universal Description Discovery and Integration. <https://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v2.htm/>, 2008.
Cité page 28.
- [161] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Greenwood, C. Goble, A. Wipat, P. Li, and T. Carver. Delivering web service coordination capability to users. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters, WWW Alt. '04*, pages 438–439, New York, NY, USA, 2004. ACM.
Cité page 39.
- [162] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing : State of the art and research challenges. *Computer*, 40(11) :38–45, Nov. 2007.
Cité page 28.
- [163] P. Parizek and F. Plasil. Modeling environment for component model checking from hierarchical architecture. *Electronic Notes in Theoretical Computer Science*, 182 :139–153, 2007.
Cité page 59.
- [164] C. Pautasso and G. Alonso. Parallel computing patterns for grid workflows. In *Workflows in Support of Large-Scale Science, 2006. WORKS'06. Workshop on*, pages 1–10. IEEE, 2006.
Cité page 79.
- [165] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10) :46–52, Oct 2003.
Cité page 28.
- [166] C. Pérez, T. Priol, and A. Ribes. A parallel corba component model for numerical code coupling. *International Journal of High Performance Computing Applications*, 17(4) :417–429, 2003.
Cité page 26.
- [167] C. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut fur instrumentelle Mathematik, Bonn, 1962.
Cité page 63.
- [168] A. Phillips and L. Cardelli. Efficient, correct simulation of biological processes in the stochastic pi-calculus. In *Computational methods in systems biology*, pages 184–199. Springer, 2007.
Cité page 62.

- [169] S. Pillai, V. Silventoinen, K. Kallio, M. Senger, S. Sobhany, J. G. Tate, S. S. Velankar, A. Golovin, K. Henrick, P. M. Rice, P. Stoehr, and R. Lopez. Soap-based services provided by the european bioinformatics institute. *Nucleic Acids Research*, 33(Web-Server-Issue) :25–28, 2005.
Cité page 39.
- [170] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell (In a Nutshell (O'Reilly))*. O'Reilly Media, Inc., 2005.
Cité page 25.
- [171] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.
Cité page 60.
- [172] I. Poupyrev and T. Ichikawa. Manipulating objects in virtual worlds : Categorization and empirical evaluation of interaction techniques. *Journal of Visual Languages & Computing*, 10(1) :19 – 35, 1999.
Cité page 15.
- [173] W. Ribarsky and J. D. Foley. Next-generation data visualization tools. *Scientific Visualization*, 1994.
Cité page 13.
- [174] M. Richards, M. Ghanem, M. Osmond, Y. Guo, and J. Hassard. Grid-based analysis of air pollution data. *Ecological Modelling*, 194(1-3) :274–286, 2006.
Cité page 37.
- [175] A. Rowe, D. Kalaitzopoulos, M. Osmond, M. Ghanem, and Y. Guo. The discovery net system for high throughput bioinformatics. In *Proceedings of the Eleventh International Conference on Intelligent Systems for Molecular Biology, June 29 - July 3, 2003, Brisbane, Australia*, pages 225–231, 2003.
Cité page 37.
- [176] J. Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.
Cité page 22.
- [177] D. Sangiorgi and D. Walker. *PI-Calculus : A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.
Cité page 62.
- [178] E. B. Shapiro. Network timetable. 1969.
Cité page 20.
- [179] M. Shaw and D. Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
Cité page 24.
- [180] J. Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1999.
Cité page 26.

- [181] J. Sifakis. Rigorous system design. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 292–292. ACM, 2014.
Cité page 31.
- [182] D. Skogan, R. Grønmo, and I. Solheim. Web service composition in uml. In *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, pages 47–57. IEEE, 2004.
Cité page 56.
- [183] R. Soley et al. Model driven architecture. *OMG white paper*, 308(308) :5, 2000.
Cité page 54.
- [184] J. Sroka, J. Hidders, P. Missier, and C. Goble. A formal semantics for the taverna 2 workflow model. *Journal of Computer and System Sciences*, 76(6) :490 – 508, 2010. Special Issue : Scientific Workflow 2009 The 2nd International Workshop on Workflow Management and Application in Grid Environments. The 3rd International Workshop on Workflow Management and Applications in Grid Environments.
Cité page 39.
- [185] J. Syed, M. Ghanem, and Y. Guo. Discovery processes : representation and reuse. In *Proceedings of First UK e-Science All-hands Conference, Sheffield, UK, 2002*.
Cité page 38.
- [186] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.
Cité page 23.
- [187] N. Sörensson and N. Een. Open-source SAT solver. <http://minisat.se/>, 2008.
Cité page 143.
- [188] A. Taherkordi, R. Rouvoy, and F. Eliassen. A component-based approach for service distribution in sensor networks. In *Proceedings of the 5th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, pages 22–28. ACM, 2010.
Cité page 30.
- [189] D. Talia. Workflow systems for science : Concepts and tools. In *ISRN Software Engineering, vol. 2013, Article ID 404525, 15 pages*, 2013.
Cité page 42.
- [190] W. Tan, Y. Fan, and M. Zhou. A petri net-based method for compatibility analysis and composition of web services in business process execution language. *Automation Science and Engineering, IEEE Transactions on*, 6(1) :94–106, 2009.
Cité page 63.
- [191] W. Tan, P. Missier, I. Foster, R. Madduri, D. De Roure, and C. Goble. A comparison of using taverna and bpel in building scientific workflows : The case of cagrid. *Concurr. Comput. : Pract. Exper.*, 22(9) :1098–1117, June 2010.
Cité page 39.

- [192] I. Taylor, M. Shields, I. Wang, and A. Harrison. The triana workflow environment : Architecture and applications. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 320–339. Springer London, 2007.
Cité page 40.
- [193] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice : The condor experience : Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4) :323–356, Feb. 2005.
Cité page 42.
- [194] J. TISSEAU. Réalité virtuelle : autonomie in virtuo. *Habilitation à Diriger des Recherches*, 2001.
Cité pages x, 15 et 16.
- [195] F. Tricas, F. Garcia-Valles, J. M. Colom, and J. Ezpeleta. A petri net structure-based deadlock prevention solution for sequential resource allocation systems. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 271–277. IEEE, 2005.
Cité page 63.
- [196] W.-T. Tsai, W. Song, R. Paul, Z. Cao, and H. Huang. Services-oriented dynamic reconfiguration framework for dependable distributed computing. *COMPSAC-NEW YORK-*, pages 554–559, 2004.
Cité page 49.
- [197] K. J. Turner and M. Sighireanu. (e)-lotos :(enhanced) language of temporal ordering specification. In *Software specification methods*, pages 165–190. Springer, 2001.
Cité page 62.
- [198] H. van der Aa, H. A. Reijers, and I. T. P. Vanderfeesten. Composing workflow activities on the basis of data-flow structures. In F. Daniel, J. Wang, and B. Weber, editors, *Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings*, volume 8094 of *Lecture Notes in Computer Science*, pages 275–282. Springer, 2013.
Cité page 37.
- [199] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages : An annotated bibliography. *Sigplan Notices*, 35(6) :26–36, 2000.
Cité page 50.
- [200] W. vanderAalst and K. vanHee. *Workflow Management : Models, Methods, and Systems*. MIT Press, Cambridge, MA, USA, 2004.
Cité page 36.
- [201] P. Vařeková and I. Černá. Model checking of control-user component-based parametrised systems. In *Component-Based Software Engineering*, pages 146–162. Springer, 2008.
Cité page 59.

- [202] V. Vyatkin. *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. ISA, 2007.
Cité page 30.
- [203] V. Vyatkin and V. Dubinin. Refactoring of execution control charts in basic function blocks of the iec 61499 standard. *Industrial Informatics, IEEE Transactions on*, 6(2) :155–165, 2010.
Cité page 31.
- [204] W3C. Web service description language. <http://www.w3.org/TR/wsdl/>, 2001.
Cité page 28.
- [205] W3C. Web Services Conversation Language. <http://www.w3.org/TR/wscl10/>, 2002.
Cité page 28.
- [206] W3C. Simple Object Access Protocol. <http://www.w3.org/2002/ws/chor/>, 2006.
Cité page 28.
- [207] W3C. Web Services Choreography Description Language. <http://www.w3.org/TR/soap/>, 2007.
Cité page 28.
- [208] A. A. Wakrime, S. Limet, and S. Robert. Réseaux fifo colorés stricts pour la formalisation des applications de visualisation scientifique interactives. *Revue des Nouvelles Technologies de l'Information*, 6ème Conférence francophone sur les Architectures Logicielles, RNTI-L-7 :103–122, 2014.
Cité pages 154 et 155.
- [209] A. A. Wakrime, S. Limet, and S. Robert. Modeling interactive scientific visualization applications with strict colored fifo nets. *Global Journal of Engineering Science and Researches*, GJESR-V-2 :36–39, 2015.
Cité pages 154 et 155.
- [210] J.-D. Warnier and B. Flanagan. *Logical construction of programs (LCP)*. Stenfort Kroese, 1974.
Cité page 20.
- [211] P. Wensch, C. van Treeck, A. Borrmann, E. Rank, and O. Wensch. Computational steering on distributed systems : Indoor comfort simulations as a case study of interactive CFD on supercomputers. *Int. J. Parallel Emerg. Distrib. Syst.*, 22(4) :275–291, 2007.
Cité page 17.
- [212] J. Wing. A specifier's introduction to formal methods. *Computer*, 23(9) :8–22, Sept 1990.
Cité page 58.
- [213] K. Wolstencroft, R. Haines, D. Fellows, A. R. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Baccall, A. Hardisty, A. N. de la Hidalga, M. P. B. Vargas, S. Sufi, and C. A. Goble.

- The taverna workflow suite : designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(Webserver-Issue) :557–561, 2013.
Cité page 39.
- [214] P. C. Wong and J. Thomas. Visual analytics. *IEEE Computer Graphics and Applications*, 24(5) :20–21, 2004.
Cité page 3.
- [215] H. Wu, F. Zhong, and Z. Duan. A π -calculus based model for web services composition. In *Pervasive Computing and Applications, 2006 1st International Symposium on*, pages 1–6. IEEE, 2006.
Cité page 62.
- [216] P. Xiong, Y. Fan, and M. Zhou. A petri net approach to analysis and composition of web services. *Systems, Man and Cybernetics, Part A : Systems and Humans, IEEE Transactions on*, 40(2) :376–387, 2010.
Cité page 63.
- [217] J. Yu, Q. Z. Sheng, and J. K. Swee. Model-driven development of adaptive service-based systems with aspects and rules. In *Web Information Systems Engineering–WISE 2010*, pages 548–563. Springer, 2010.
Cité page 50.
- [218] Y. Yu, T. Li, Q. Liu, and F. Dai. Approach to modeling components in software architecture. *Journal of Software*, 6(11) :2196–2200, 2011.
Cité pages 63 et 89.
- [219] M. Zain-ul Abdein, D. Nelias, J.-F. Jullien, and D. Deloison. Prediction of laser beam welding-induced distortions and residual stresses by numerical simulation for aeronautic application. *Journal of Materials Processing Technology*, 209(6) :2907–2917, 2009.
Cité page 1.
- [220] D. Zeltzer. Autonomy, interaction, and presence. *Presence : Teleoper. Virtual Environ.*, 1(1) :127–132, Jan. 1992.
Cité page 15.
- [221] H. Zeng and H. Miao. Deadlock detection for parallel composition of components. In *Computer and Information Science 2010*, pages 23–34. Springer, 2010.
Cité page 4.
- [222] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.
Cité page 61.

Résumé. Les architectures par composants sont de plus en plus étudiées et utilisées pour le développement efficace des applications en génie logiciel. Elles offrent, d'un côté, une architecture claire aux développeurs, et de l'autre, une séparation des différentes parties fonctionnelles et en particulier dans les applications de visualisation scientifique interactives. La modélisation de ces applications doit permettre la description des comportements de chaque composant et les actions globales du système. De plus, les interactions entre composants s'expriment par des schémas de communication qui peuvent être très complexes avec, par exemple, la possibilité de perdre des messages pour gagner en performance. Cette thèse décrit le modèle ComSA (Component-based approach for Scientific Applications) qui est basé sur une approche par composants dédiée aux applications de visualisation scientifique interactive et dynamique formalisée par les réseaux FIFO colorés stricts (sCFN). Les principales contributions de cette thèse sont dans un premier temps, un ensemble d'outils pour modéliser les différents comportements des composants ainsi que les différentes politiques de communication au sein de l'application. Dans un second temps, la définition de propriétés garantissant un démarrage propre de l'application en analysant et détectant les blocages. Cela permet de garantir la vivacité tout au long de l'exécution de l'application. Finalement l'étude de la reconfiguration dynamique des applications d'analyse visuelle par ajout ou suppression à la volée d'un composant sans arrêter toute l'application. Cette reconfiguration permet de minimiser le nombre de services non disponibles.

Mots clés. Architectures par composants, Visualisation Scientifique, Vérification, Reconfiguration Dynamique, Analyse Visuelle, Réseaux FIFO Colorés stricts, Vivacité, Applications Interactives Distribuées/Parallèles.

A component-based approach for interactive visual analysis of numerical simulation results

Abstract. Component-based approaches are increasingly studied and used for the effective development of the applications in software engineering. They offer, on the one hand, safe architecture to developers, and on the other one, a separation of the various functional parts and particularly in the interactive scientific visualization applications. Modeling such applications enables the behavior description of each component and the global system's actions. Moreover, the interactions between components are expressed through a communication schemes sometimes very complex with, for example, the possibility to lose messages to enhance performance. This thesis describes ComSA model (Component-based approach for Scientific Applications) that relies on a component-based approach dedicated to interactive and dynamic scientific visualization applications and its formalization in strict Colored FIFO Nets (sCFN). The main contributions of this thesis are, first, the definition of a set of tools to model the component's behaviors and the various application communication policies. Second, providing some properties on the application to guarantee it starts properly. It is done by analyzing and detecting deadlocks. This ensures the liveness throughout the application execution. Finally, we present dynamic reconfiguration of visual analytics applications by adding or removing on the fly of a component without stopping the whole application. This reconfiguration minimizes the number of unavailable services.

Keywords. Component-based approaches, Scientific Visualization, Verification, Dynamic Reconfiguration, Visual Analytics, strict Colored FIFO Nets, Liveness, Distributed/Parallel Interactive Applications.